Testaus Scrum-prosessimallissa

Mikko Pöri

Helsinki 11.9.2008

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Mikko Pöri

Työn nimi — Arbetets titel — Title
Testaus Scrum-prosessimallissa

Oppiaine — Läroämne — Subject
Ohjelmistotekniikka

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	11.9.2008	63 sivua + 14 liitesivua

Tiivistelmä — Referat — Abstract

Testaus ketterissä menetelmissä (agile) on kirjallisuudessa heikosti määritelty, ja yritykset toteuttavat laatu- ja testauskäytäntöjä vaihtelevasti. Tämän tutkielman tavoitteena oli löytää malli testauksen järjestämiseen ketterissä menetelmissä. Tavoitetta lähestyttiin keräämällä kirjallisista lähteistä kokemuksia, vaihtoehtoja ja malleja. Löydettyjä tietoja verrattiin ohjelmistoyritysten käytännön ratkaisuihin ja näkemyksiin, joita saatiin suorittamalla kyselytutkimus kahdessa Scrum-prosessimallia käyttävässä ohjelmistoyrityksessä.

Kirjallisuuskatsauksessa selvisi, että laatusuunnitelman ja testausstrategian avulla voidaan tunnistaa kussakin kontekstissa tarvittavat testausmenetelmät. Menetelmiä kannattaa tarkastella ja suunnitella iteratiivisten prosessien aikajänteiden (sydämenlyönti, iteraatio, julkaisu ja strateginen) avulla.

Tutkimuksen suurin löytö oli, että yrityksiltä puuttui laajempi ja suunnitelmallinen näkemys testauksen ja laadun kehittämiseen. Uusien laatu- ja testaustoimenpiteiden tarvetta ei analysoitu järjestelmällisesti, olemassa olevien käyttöä ei kehitetty pitkäjänteisesti, eikä yrityksillä ollut kokonaiskuvaa tarvittavien toimenpiteiden keskinäisistä suhteista. Lisäksi tutkimuksessa selvisi, etteivät tiimit kyenneet ottamaan vastuuta laadusta, koska laatuun liittyviä toimenpiteitä tehdään iteraatioissa liian vähän. Myös Scrum-prosessimallin noudattamisessa oli korjaamisen varaa. Yritykset kuitenkin osoittivat halua ja kykyä kehittää toimintaansa ongelmien tunnistamisen jälkeen.

ACM Computing Classification System (CCS 1998):
D.2.5 Testing and Debugging
D.2.9 Management
K.6.1 Project and People Management
K.6.3 Software Management

Avainsanat — Nyckelord — Keywords
testaus, laatu, ketterät prosessimallit, agile, XP, Scrum, TDD, ATDD, BDD

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpulan tiedekirjasto, sarjanumero C-

Muita tietoja — övriga uppgifter — Additional information

Saatteeksi

Tuntuu helpottavalta kirjoittaa tämän pro gradu -työni viimeistä sivua. Olen saavutukseeni tyytyväinen ja haluan nöyrimmästi kiittää kaikkia niitä, jotka ovat minua tielläni auttaneet.

Gradun kommentoinnista, neuvoista ja parannusideoista haluan kiittää Juhani Snellmania, Sampo Lehtistä, Joni Karppista, Katja Kuusikumpua, Maaret Pyhäjärveä, Antti Heimolaa, Eetu Huismania ja Matti Paksulaa. Neuvoista yritysten valitsemisen ja lähestymisen suhteen olen kiitollinen Lasse Koskelalle ja Mikko Raidalle. Avusta materiaalien ja kommenttien muodossa kiitän Juha Itkosta. Kiitän myös yritysten A ja B henkilökuntaa kärsivällisyydestä suhteeni ja rakentavasta asennoitumisesta kyselyyn.

Lukemisesta ja arvokkaista kommenteista työn kokonaisuuden suhteen olen kiitollinen äidilleni Liisa Pörille, isälleni Keijo Pörille sekä opiskelutovereilleni Sebastian Siikavirralle, Mikko Warikselle ja Jesse Liukkoselle. Te autoitte luomaan tästä eheän kokonaisuuden.

Jatkuvasta ohjauksesta ja tuesta työnantajallani Efectellä kiitän Kai Virkkiä. Ohjauksesta Tietojenkäsittelytieteen laitoksella kiitän professori Juha Tainaa ja vanhempaa lehtoria Teemu Kerolaa. Olette jaksaneet vastaaninttävää opiskelijaa pedagogisella kärsivällisyydellä.

Loppusilauksen työlleni antoivat kauniimman suomen kielen osalta Mari Nikonen sekä kauniiden kuvien osalta hyvä ystäväni Tuomas Rentola. Kiitos ja kumarrus.

Henkisestä tuesta koko opiskelujeni aikana omistan pro gradu -työni äidilleni Liisa Pörille.

Helsingissä 11.9.2008

Mikko Pöri

Sisältö

Liitteet

1 Johdanto

Testaus osoittaa ohjelmiston toimivan oikein tai väärin suhteessa eksplisiittisiin tai implisiittisiin vaatimuksiin. Eksplisiittiset vaatimukset on kirjallisesti kuvattu ohjelmiston vaatimusmäärittelyssä, ja testauksen tulee selvittää, täyttyväkö nämä vaatimukset. Implisiittiset vaatimukset ovat kirjallisesti määrittelemättömiä vaatimuksia, joita ohjelmiston tilaaja tai käyttäjät olettavat ohjelmiston joka tapauksessa toteuttavan. Tämän vuoksi implisiittisten vaatimusten testaus on vaikeampaa.

Testausta voi tapahtua jokaisessa ohjelmistotuotantoprosessin vaiheessa tai sen jälkeen. Osana ohjelmistotuotantoprosessia testaus auttaa parantamaan ohjelmiston laatua ja oikeellisuutta sekä vähentää sen tuotanto- ja ylläpitokustannuksia [Pre97]. Ohjelmistotuotantoprosessin jälkeen tilaaja voi testauksella mitata ohjelmiston laatua ja oikeellisuutta. Ohjelmiston ja sen toimintaympäristön ominaisuudet vaikuttavat testauksen tarpeisiin.

Scrum ja muut ketterät (agile) prosessimallit ovat nopeasti yleistyneet ohjelmistotuotannossa. Ohjelmistotuotannon prosessimallin tulee asettaa raamit testauksen suunnittelulle. Kuitenkaan ketterät prosessimallit eivät selkeästi määrittele, kuinka testaus tulee toteuttaa eikä testauksen tutkimuksen tuloksia riittävästi huomioida. Myöskään pitkään tunnettuihin testauksen haasteisiin ei vastata. Testauskäytännöt ovat sekoitus uutta ja vanhaa; sekä ennen julkaisua tehtävää perinteistä järjestelmätestausta että Test Driven Development -menetelmän mukaista yksikkötestauksen korostamista.

Epätäydellinen prosessimalli saattaa johtaa siihen, että kehittäjät jättävät oleellisia laadunvarmistuksen toimenpiteitä suorittamatta. Jos laadunvarmistuksen toimenpiteitä jätetään tekemättä, päätöksen tulee perustua täyteen tietoisuuteen sen vaikutuksista ohjelmiston ja tuotantoprosessin laatuun sekä mahdollisesti koko yrityksen liiketoimintaan.

Tämän tutkielman tavoitteena oli löytää malli testauksen järjestämiseen ketterissä menetelmissä. Tavoitetta lähestyttiin keräämällä kirjallisista lähteistä kokemuksia, vaihtoehtoja ja malleja. Löydettyjä tietoja verrattiin ohjelmistoyritysten käytännön ratkaisuihin ja näkemyksiin, joita saatiin suorittamalla kyselytutkimus kahdessa Scrum-prosessimallia käyttävässä ohjelmistoyrityksessä. Kyselytutkimuksessa haastateltiin testausta ja tuotekehitystä tekeviä, suunnittelevia ja johtavia henkilöitä. Tutkimukseen haettiin yrityksiä, jotka sijaitsevat pääkaupunkiseudulla ja joilla on omaa tuotekehitystä. Lisäksi edellytettiin, että yritykset työllistävät 40-100 henkeä

ja että niillä on ainakin muutaman vuoden kokemus Scrumin käytöstä.

Tutkimuksessa löytyi huomattavia puutteita yritysten tavoissa järjestää testausta. Testausta ei suunniteltu strategisesti osana ohjelmiston laatutavoitteita eikä laatutavoitteita määritelty. Manuaalinen ja automatisoitu testaus oli yksipuolisesti ja puutteellisesti toteutettu. Prosessimallin noudattamisesta ja toiminnan kehittämisestä ei huolehdittu.

Tämän tutkielman rakenne on seuraava. Luvussa 2 esitellään yleisimmät ja ketterien prosessimallien testauksessa tärkeimmät testausmenetelmät. Luvussa 3 esitellään ketterien prosessimallien yleinen filosofia ja prosessimalleista yksityiskohtaisesti Extreme Programming (XP) ja Scrum, sekä kuvataan ketterien prosessimallien testauksen toimenpiteet ja suunnittelussa huomioitavat asiat. Luvussa 4 esitellään tutkimuksen tausta, yritysten valintaan vaikuttaneet tekijät ja yritysten ominaisuudet. Luvussa 5 käydään läpi haastattelujen ja muun tiedonhankinnan järjestelyt sekä toteutuminen. Löydetyt tulokset esitellään luvussa 6. Luvussa 7 esitellään johtopäätökset tuloksista. Luku 8 on yhteenveto.

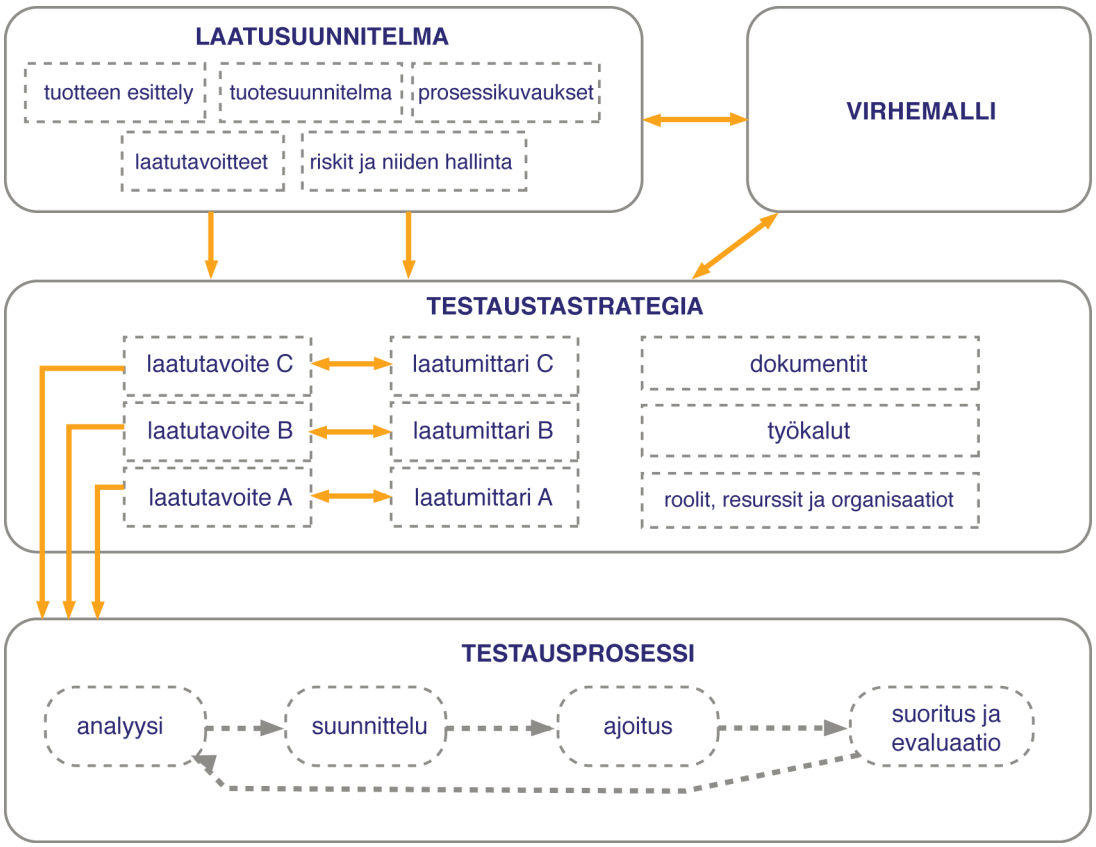
2 Ohjelmistojen testaus

Ohjelmiston testaus toteutetaan testausprosessin kautta *laatusuunnitelman* ja siitä johdetun *testausstrategian* pohjalta. Laatusuunnitelman, testausstrategian ja testausprosessin keskinäiset suhteet ja rakenne on esitetty kuvassa 1. Laatusuunnitelma määrittelee tavoitteet ohjelmiston laadulle ja keinot tavoitteiden saavuttamiseksi. Laatusuunnitelma koostuu viidestä osasta. *Tuotteen esittelyssä* määritellään lyhyesti tuotteen ominaisuudet, markkinat ja kilpailutilanteen edellytykset laadulle. *Tuotesuunnitelma* pitää sisällään julkaisujen päivämäärät, vastuut, tarkistuspisteet sekä jakelun ja ylläpidon suunnitelmat. *Prosessikuvaukset* koostuvat tuotteen kehittämisen ja ylläpidon prosessien kuvauksista. *Laatutavoitteet* määrittelevät tavoitteet sekä kehittämissuunnitelmat ja -keinot laadulle. Osana laatutavoitteita luokitellaan tuotteen kriittisimmät laatuominaisuudet. *Riskeissä ja niiden hallinnassa* kuvataan tuotteen laadun suurimmat riskit sekä toimenpiteet kunkin riskin pienentämiseksi. Laatusuunnitelman laatimisen apuna voidaan käyttää analyysejä järjestelmästä aiemmin löydetyistä virheistä ja asiakkaiden lähettämistä tukipyynnöistä. [Hum89, s. 357][Som07, s. 652]

Kullekin tuotteelle laaditaan yksilöllinen testausstrategia, joka perustuu laatusuunnitelmassa kuvattuihin laadun tavoitteisiin, keinoihin ja riskeihin. Testausstrategiat voivat vaihdella muun muassa organisaatioiden ja sovellusalueiden erojen sekä valitun prosessimallin johdosta. Yrityksen tuottaessa samalla organisaatiolla ja prosessimallilla useita tuotteita tai tuoteperhettä näiden tuotteiden testausstrategiat ovat todennäköisesti samankaltaisia ja yhtä strategiaa voidaan käyttää apuna luotaessa toisia. Testausstrategia kuvaa tarkasti ja yksiselitteisesti laadun tavoitteet ja antaa selkeän tavan mitata niiden toteutumista. Apuna tavoitteiden toteutumisen mittauksessa voidaan käyttää kattavuusanalyysiä. [PeY07, s. 377-381, 458-459]

Testausstrategian laatimista voidaan helpottaa määrittelemällä tuotteelle virhemalli eli olettamus virheiden vakavuudesta ja sijainnista. Virhemallin avulla voidaan määrittää testimalli, joka kuvaa karkealla tasolla ohjelmiston rakenteen, toiminnan ja kohteet testaukselle. Testimalli auttaa määrittämään tavoitteiden välisen tärkeysjärjestyksen ja kunkin tavoitteen toteuttamiseen vaadittavien keinojen määrän. [Bin00, s. 51]

Testausstrategia kuvaa tuotteen laadun varmistuksessa tarvittavat dokumentit ja käytetyt työkalut. Lisäksi määritellään testausstrategian toteuttamisesta vastaavien tekijöiden tai organisaatioiden roolit, vastuut, oikeudet ja resurssit. Toteuttaminen



Kuva 1: Laatusuunnitelma, testausstrategia ja testausprosessi.

voidaan osittain tai kokonaan ulkoistaa toiselle osastolle tai organisaatiolle esimerkiksi alihankintana. [PeY07, s. 377-381, 458-459]

Kukin testausstrategiassa kuvattu tavoite toteutuu käytännössä *testausprosessin* neljän osavaiheen kautta. *Analyysissa* selvitetään testauksen tavoite. Analyysin ensimmäisessä vaiheessa käydään läpi lähdedokumentaatio, joka yleensä koostuu ohjelmiston vaatimuksista ja toiminnallisista suunnitelmista. Lisäksi käytetään käyttöoppaita, kokouspöytäkirjoja ja muita hyödyllisiä lähteitä. Toisessa vaiheessa ohjelmisto jaetaan toimintoihin, jotka luokitellaan liiketoimintariskin avulla. Jaon aste riippuu testauksen asteesta eli onko kyseessä yksikkö- vai järjestelmätason testaus. Toiminnot toimivat jatkossa toiminnallisen testauksen kohteina. Kullekin toiminnolle määritetään testausehdot, jotka toteuttaessaan toiminto täyttää tarkoituksensa. Analyysin lopuksi testausehtoihin ja toimintoihin merkitään niiden yhteydet lähdedokumentaatioon, jotta jäljitettävyys ja onnistumisen arviointi helpottuvat. Analyysivaiheessa selvitetään testauksen lähestymistapa, resurssit ja rajoitteet, aikataulu sekä seuraavien vaiheiden tehtävät ja riskit. [Dav00]

Suunnittelussa määritellään testitapauksia vastaamaan analyysissa määritettyjä toimintoja ja testausehtoja. Näitä täydennetään koodin analysoinnin, heuristiikan ja testaajan muodostamien epäilyjen avulla. Lopuksi kullekin testitapaukselle määritetään odotettu tulos ja kattavuuskriteerit. *Ajoituksessa* testitapauksien keskinäiset riippuvuudet ja prioriteetit määrittävät niiden järkevän suoritusjärjestyksen. *Suorituksessa ja evaluoinnissa* katsotaan, täyttyvätkö testitapauksille suunnittelussa asetetut kattavuuskriteerit eli onko ohjelmistoa testattu tarpeeksi kattavasti. [Dav00]

Testausta voidaan toteuttaa ketteriin menetelmiin soveltuen iteratiivisesti. Tällöin valitaan osa laatutavoitteesta ja käydään läpi kaikki vaiheet analyysista suoritukseen ja evaluointiin. Lopulta palataan alkuun ja hyödyntäen edellisen iteraation kokemuksia toteutetaan seuraava osa laatutavoitteesta.

2.1 Kattavuusanalyysi

Kattavuus (coverage) antaa testaukselle laadullisen ja määrällisen mitan. Kattavuuden avulla voidaan mitata testausstrategian laatutavoitteiden toteutumista. Kattavuus kertoo, kuinka suuri osa ohjelmistosta on testauksen aikana suoritettu [Bei90, s. 14]. Kattavuus toimii pääsääntöisesti valkolaatikkotestauksen arvioinnin apuna ja kertoo suhdeluvun ohjelmiston sisäisen toiminnan yksityiskohdista. Esimerkiksi lausekattavuus kertoo, kuinka monella ohjelmiston rivillä on testiä suoritettaessa käyty. Myös mustalaatikkotestausta voidaan arvioida kattavuudella. Esimerkkinä vaatimuskattavuus, jolloin arvioidaan, kuinka suuri määrä ohjelmistolle asetetuista vaatimuksista on käyty läpi testauksen aikana [KLV05]. Pelkästään *vastuuanalyysia (responsibility analysis)* käyttämällä laaditut testausstrategiat eivät kykene saavuttamaan korkeaa kattavuustulosta [Gra92, HoM96]. Tällöin on käytettävä apuna kattavuusanalyysia, jotta testausstrategia saadaan kehitettyä aiempaa tarkemmaksi ja paremmaksi [Bin00, s. 357-401].

Testejä tai testausstrategiaa ei voida arvioida ainoastaan kattavuuden avulla, sillä se ei yksiselitteisesti mittaa testauksen tehokkuutta. Weyukerin hypoteesin mukaan kattavuusehtojen riittävyys voidaan todeta ainoastaan intuitiivisesti [Wey88, FrW93]. Oikein käytettynä hyvin valittu kattavuusehto parantaa testausta tarjoten tärkeää tietoa testien toiminnasta. Sivutuotteena kattavuuden käytössä saadaan testattavan ohjelmiston rakenteesta ja toiminnasta lisää tietoa. Lisäksi kattavuusraportit saattavat paljastaa niin sanottua kuollutta koodia eli sellaista koodia, joka ei millään ohjelmiston syötteillä voi tulla suoritetuksi.

Kaner listaa 101 erilaista tapaa mitata testauksen kattavuutta [Kan95]. Oikean kattavuuslajin tai -lajien valinta on tapauskohtaista. Valinta riippuu ohjelmiston ominaisuuksista, kuten rakenteesta ja käyttötarkoituksesta. Lisäksi tuotannossa ja testauksessa käytettävät työkalut asettavat omat vaatimuksensa yhteensopivuudesta. Valintaan vaikuttaa myös käytetty prosessimalli, eli eri rooleissa toimivilla ihmisillä on yksilölliset tiedontarpeet testauksen kattavuudesta. Kyse on siitä, mitä halutaan mitata ja millä tavoin mittauksen tulosten halutaan vaikuttavan testausprosessiin.

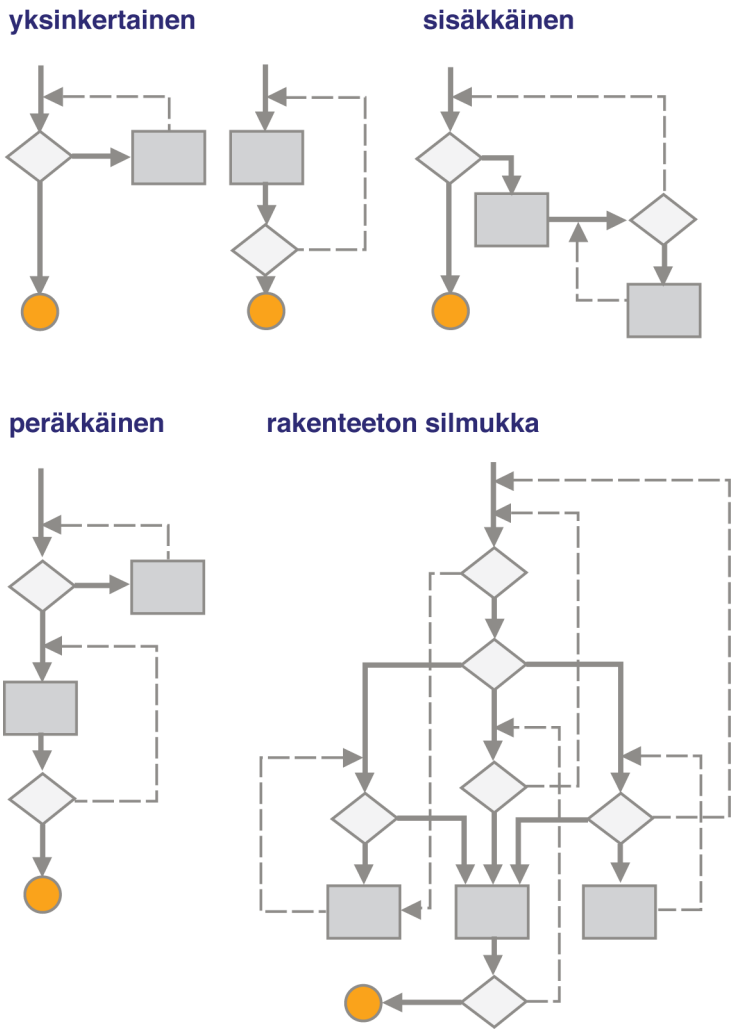
Seuraavaksi esitellään yleisimmin käytetyt kattavuudet, joita ovat lausekattavuus, haaraumakattavuus, luokkakattavuus, ehtokattavuus, moniehtokattavuus, polkukattavuus ja peruspolkukattavuus.

Lausekattavuus (statement coverage) mittaa kuinka moni ohjelmiston lauseista on suoritettu testauksen aikana. Kattavuusehtona lausekattavuus on yleensä liian löysä. Hieman tarkempi kattavuusmitta saadaan *haaraumakattavuudella (branch coverage)*, joka mittaa, kuinka moni suorituksen haarauma käydään lävitse testauksen aikana. Ei siis riitä, että lause suoritetaan, vaan suoritusta edeltäneet haaraumat ovat myös tärkeitä. *Luokkakattavuus (class coverage)* mittaa, kuinka monessa ohjelmiston luokista on käyty testauksen aikana. [Bin00, s. 357-401]

Ehtokattavuudessa (condition coverage) tutkitaan haaraumien atomisten ehtojen toteutumista. Täyden ehtokattavuuden saa sillä, että ohjelmiston jokaisen ehtolauseen jokainen atominen ehto on itsenäisesti toteutunut. Tästä hienojakoisempi mitta on *moniehtokattavuus (multicondition coverage)*, joka mittaa kaikkien atomisten ehtolauseiden välisten kombinaatioiden toteutumista. [Bin00, s. 357-401]

Silmukkakattavuus (covering iteration) mittaa ohjelmiston silmukoiden suorituspolkujen läpikäyntiä. Erilaisille silmukkatyypeille (yksinkertaiset, sisäkkäiset, peräkkäiset ja rakenteettomat silmukat) on erilaisia läpikäyntivaatimuksia. Silmukkatyypit on esitelty kuvassa 2. Silmukoiden kaikkien polkujen läpikäynti ei ole polkujen mahdollisen suuren määrän vuoksi järkevää tai edes mahdollista, minkä vuoksi niille määritellään silmukan tyyppiin liittyvät lineaariset läpikäyntivaatimukset. [Bin00, s. 357-401]

Polkukattavuus (path coverage) mitataan käymällä läpi kaikki ohjelmiston mahdolliset suorituspolut. Suorituspolku on lauseiden ja siirtymien kautta kulkeva reitti ohjelmiston alusta loppuun. Polkuja voi olla äärettömästi tai ainakin hyvin iso määrä, minkä vuoksi polkukattavuutta ei käytännössä voi käyttää. Esimerkiksi yksinkertaisessa koodissa, jossa on kymmenen peräkkäistä if-lausetta, on mahdollisia polkuja 2^{10} eli 1024 kappaletta. Käytännöllisempi muunnelma polkukattavuudesta on



Kuva 2: Esimerkkejä erilaisista silmukoista vasemmalta ylhäältä lukien: yksinkertainen, sisäkkäinen, peräkkäinen ja rakenteeton silmukka [Bin00, s. 358].

peruspolkukattavuus (basic-path coverage), joka mittaa toisistaan riippumattomien polkujen läpikäyntiä. Riippumaton polku on sellainen, jota ei saada lineaarisena kombinaationa muista riippumattomista poluista. [Bin00, s. 357-401]

Kattavuuksia ei voida verrata toisiinsa yksiselitteisesti, mutta joitain asioita voidaan analyyttisesti todeta. Lausekattavuus sisältyy haaraumakattavuuteen, sillä jokaisen haarauman suorittaminen johtaa väistämättä jokaisen lauseen suorittamiseen. Polkukattavuus sisältää useita muita suorituspolkuihin perustuvia kattavuuksia, sillä se sisältää ohjelmiston kaikki mahdolliset suorituspolut.

Ohjelmiston eri osien testauksen välillä on prioriteettieroja. Jotkin kohdat voivat olla kriittisempiä ohjelmiston toiminnan kannalta tai ne on tilastollisesti havaittu erittäin

virhealttiiksi. Testauksen Pareto-Zipf -laki vahvistaa saman: 80 % virheistä löytyy 20 % moduuleista [Par97, Wei71]. Kattavuustavoitteet kriittisten ja virhealttiiden osien kohdalla tulee asettaa korkeammiksi [Mar99].

Kattavuustyökalulla saatu tulos ei saa ohjata testien kirjoittamista, vaan tuloksen tulee parantaa testausprosessia. Kun kattavuusanalyysissa huomataan, että jokin koodin osa on jäänyt suorittamatta testauksen aikana, testejä on helppo korjata suorittamaan puuttuva osa. Tärkeintä on kuitenkin löytää virheet testauksen suunnittelussa. Korjaamalla suunnittelun puutteet ja korjaamalla testit vastaamaan suunnitelmaa kattavuuskriteerit tulevat aikanaan täytetyiksi. Tavoitteena ei ole kirjoittaa testejä, jotka vain yhdellä kattavuuslajilla mitattuna saavuttavat mahdollisimman hyvän tuloksen. Tavoitteena on luoda systemaattinen testausstrategia, joka löytää mahdollisimman monta virhettä ohjelmistosta. [Mar99]

2.2 Testausmenetelmien luokittelu

Testausmenetelmiä luokitellaan monilla eri tavoilla. Yleisimmin ne jaotellaan toiminnalliseen ja ei-toiminnalliseen testaukseen, musta-, harmaa- ja valkolaatikkotestaukseen, automatisoituun ja manuaalisen testaukseen, positiiviseen ja negatiiviseen testaukseen sekä tutkivaan ja varmentavaan testaukseen [KaL00, Mye04] [PeY07, s. 161-162] [Bei90, s. 535, 539] [Kan04]. Uusimpana tulokkaana on ketterien menetelmien mukainen luokittelu synkronoituun, osittain synkronoituun ja synkronoimattomaan testaukseen [Itk07].

2.2.1 Musta-, harmaa- ja valkolaatikkotestaus

Mustalaatikkotestauksessa (black-box testing) testataan ohjelmiston toiminnallisuutta tuntematta ohjelmiston sisäisiä rakenteita. Tällöin testausta suunnataan käyttäen apuna ohjelmistolle vaatimusmäärittelyssä muodostettuja vaatimuksia tai muuta saatavilla olevaa tietoa ohjelmiston toiminnasta. Mustalaatikkotestaus suoritetaan käyttöliittymän tai muun rajapinnan kautta. *Harmaalaatikkotestaus (grey-box testing)* on muuten samanlainen kuin mustalaatikkotestaus, mutta harmaalaatikkotestauksessa testaaja tuntee jonkin verran ohjelmiston sisäistä toiminnallisuutta ja saa alustaa tai muokata testausympäristöä, kuten asettaa tietokantaan tiettyjä arvoja. *Valkolaatikkotestauksessa (white-box testing, lasilaatikkotestaus, glass-box testing)* testaaja tuntee ohjelmiston sisäisen rakenteen lähdekoodeineen. Valkolaatikkotestaus kohdistuu ohjelmiston rakenteeseen eli metodien, funktioiden tai moduulien si-

säisiin ja keskinäisiin toimintoihin. Lisäksi valkolaatikkotestaaja käyttää sisäisen rakenteen tuntemustaan ohjelmiston testaukseen sen ulkopuolelta sopivan rajapinnan, kuten olio- tai komponenttirajapinnan tai käyttöliittymän, avulla. Testausmenetelmiä ei voida ennalta täysin luokitella valko-, harmaa- tai mustalaatikkotestaukseksi, vaikka esimerkiksi yksikkötestit ovat usein valkolaatikkotestejä ja beeta-testaus mustalaatikkotestausta. Tarkka luokittelu riippuu yksittäistapauksesta. [KaL00, Mye04]

2.2.2 Toiminnallinen ja ei-toiminnallinen testaus

Toiminnallinen testaus *(functional testing)* tutkii ohjelmiston toiminnallisten vaatimusten toteutumista. Toiminnalliset vaatimukset toteuttavat ohjelmistoon kohdistettuja käyttötapauksia, kuten tietyn käyttäjän suorittaman toimintosarjan toteutumista. Ei-toiminnallinen testaus *(non-functional testing)* varmistaa ei-toiminnallisten vaatimusten toteutumista. Ei-toiminnalliset vaatimukset määrittävät järjestelmän toiminnalle reunaehtoja ja laatuominaisuuksia. Esimerkkeinä ei-toiminnallisista vaatimuksista ovat suorituskyky-, turvallisuus-, laajennettavuus-, ja ylläpidettävyysvaatimukset. Ei-toiminnalliset vaatimukset tukevat toiminnallisia vaatimuksia ja kuvaavat koko järjestelmän toimintaa.

Toiminnallinen ja ei-toiminnallinen testaus eivät johda testitapauksia ohjelmiston rakenteesta tai toteutuksesta. Tämän vuoksi toiminnallisten ja ei-toiminnallisten testitapausten suunnittelu voidaan aloittaa samaan aikaan vaatimusmäärittelyprosessin kanssa. Toiminnallista ja ei-toiminnallista testausta kutsutaan myös vaatimuslähtöiseksi testaukseksi *(specification-based testing)*, ja se on lähtökohtaisesti mustalaatikkotestausta. [PeY07, s. 161-162]

2.2.3 Negatiivinen ja positiivinen testaus

Negatiivinen testaus pyrkii osoittamaan, ettei testattava ohjelmisto toimi [Bei90, s. 535]. Tämän vuoksi negatiivinen testaus etsii virheitä, jotka aiheuttavat merkittävän ongelman järjestelmän toiminnalle, kuten järjestelmän kaatumisen, tiedon korruptoitumisen tai tietoturvariskin. Lisäksi negatiivisella testauksella saadaan havaintoja ja mittaustuloksia järjestelmän reagoinnista ulkoisiin ongelmatilanteisiin sekä etsitään ja tunnistetaan ohjelmistosta väärinkäytöksille alttiita osia. Edellä mainittujen tärkeimpien tavoitteiden lisäksi negatiivinen testaus voi testata vahingollisten käyttäjäsyötteiden vaikutusta, sisäisen datan validoinnin ja hylkäämisen seurauksia, ohjelmiston selviytymistä puuttuvilla ja huonoilla ulkoisilla resursseilla, virheiden

käsittelyä ja virhetilanteista palautumista. Negatiivinen testaus käyttää usein testausmenetelmänä tutkimusmatkailevaa testausta eli intuitioon perustuvaa kokeilevaa testausmenetelmää [Lyn03]. Negatiivisen testauksen vastakohta on *positiivinen testaus*, jonka tavoitteena on osoittaa ohjelmiston toimivan määritysten mukaisesti [Bei90, s. 539]. Testausmenetelmiä ei voida suoraan luokitella näihin kahteen kategoriaan, vaikka esimerkiksi tutkimusmatkaileva testaus on usein negatiivista testausta ja automatisoitu suorituskykytestaus usein positiivista testausta. Tarkka luokittelu riippuu yksittäistapauksesta.

2.2.4 Tutkiva ja varmentava testaus

Varmentava testaus *(confirmatory testing)* pyrkii pitämään ohjelmiston olemassa olevat ominaisuudet kunnossa, vaikka uusia ominaisuuksia jatkuvasti lisätään [Kan04]. Varmentavassa testauksessa kunkin testin oikea lopputulos on tiedossa etukäteen. Ketterissä menetelmissä varmentava testaus on automatisoitua ja tuotetaan osana ohjelmistopiirteiden kehitystä. Tutkivassa testauksessa *(investigative testing)* testin lopputulos ei ole ennalta tiedossa ja siinä, kuten negatiivisessakin testauksessa, etsitään ennalta arvaamattomia virheitä järjestelmän toiminnasta. Tutkiva testaus käyttää usein testausmenetelmänä tutkimusmatkailevaa *(exploratory testing)* testausta. Ketterissä menetelmissä tutkiva testaus on usein manuaalista ja tutkimusmatkailevaa testausta. Jako tutkivaan ja varmentavaan testaukseen muistuttaa läheisesti jakoa positiiviseen ja negatiiviseen testaukseen.

2.2.5 Automatisoitu ja manuaalinen testaus

Automatisoidun testauksen vastakohta on *manuaalinen testaus*, jossa ohjelmiston virheitä etsitään suorittamalla testitapauksia käsin. *Automatisoidussa testauksessa* testitapaukset on automatisointityökalun tai kirjaston avulla ohjelmoitu tai tallennettu muistiin ja ovat toistuvasti suoritettavissa.

Testauksen automatisointi nopeuttaa virheenkorjausta ja onnistuneen korjauksen varmistamista. Automatisoinnin avulla voidaan helposti tuottaa säännöllisiä raportteja testauksesta. Automatisoidut testit vähentävät organisaation riippuvuutta taitavista testaajista ja pienentävät inhimillisten virheiden riskiä. Automatisoitujen testien ajaminen on edullista, eli automatisointi lisää testauksen tuottavuutta. Lisäksi automatisointi mahdollistaa pitkien ja monimutkaisten järjestelmätason testien suorittamisen ja suuren testisyöteaineiston käytön [Bin00, s. 801-806]. Automa-

tisoinnin haittapuolia ovat testien kirjoittamisen ja ylläpidon kustannukset. Automatisoidun testauksen onnistumista auttaa järkevän testausstrategian laatiminen, jolloin on selkeää, kuinka paljon automatisointia tarvitaan milläkin testauksen osa-alueella. Testattavan ohjelmiston rakennetta tulee muuttaa testauksen vaatimusten mukaan *(Design for Testability)*, jotta testien kirjoittaminen helpottuu. Käytettäessä jatkuvaa integraatiota automatisoidut testit pysyvät ajan tasalla ja niissä ilmenneet virheet ja puutteet on tehokkain korjata. Automatisoidut testit ovat ohjelmiston osia siinä missä tuotantokoodikin, eli samojen hyvien ohjelmistokehitysstandardien noudattaminen helpottaa niiden ylläpitoa [BWK05, Kan97].

Tehtäessä valintaa manuaalisen ja automatisoidun testauksen välillä on oleellista miettiä kummankin lähestymistavan hyötyjä. Eräs yksinkertainen tapa mitata automatisoinnin arvoa on laskea, kuinka monta manuaalista testiä sen hinnalla olisi voitu tehdä ja mitä virheitä testit olisivat löytäneet [Mar99]. On kuitenkin esitetty, että "hyvää manuaalista testiä ei voi automatisoida"[Bac06]. Perusteena ovat hyvien automatisoitujen testien ja hyvien manuaalisten testien erilaiset vahvuudet virheiden löytämisessä. Automatisoitu testi on helposti ja nopeasti toistettavissa; niitä voidaan suorittaa jatkuvasti projektin taustalla lähes ilmaiseksi. Automatisoiduilla testeillä voidaan rakentaa monimutkaisia testikokonaisuuksia, joilla ohjelmistoa voidaan kokeilla ja rasittaa monipuolisesti. Manuaalisen testin vahvuus on ihmisen kyky tunnistaa intuitiolla, mielikuvituksella, kokemuksella ja loogisella päättelyllä virhe- ja ongelmakohtia. Esityksen korollaarit kuuluvat seuraavasti: "jos manuaalinen testi on hyvin automatisoitavissa, ei se alun perin ollut hyvä manuaalinen testi"ja "erinomainen automatisoitu testi ei ole sama kuin se manuaalinen testi, jonka luulit automatisoineesi"[Bac06].

Manuaalinen ja automatisoitu testaus löytävät pääosin erityyppisiä virheitä, joten niiden suora vertailu ei ole järkevää. Testauksen automatisoinnin kannattavuutta arvioitaessa tulee huomioida kummankin testausmuodon kokonaiskustannukset ja erilaiset hyödyt. Hyötyjä punnitessa tulee huomioida eri testien välisten prioriteettierojen merkitys, ohjelmistoprojektin konteksti ja mahdolliset piilotetut lisäkulut. Lisäkuluja muodostuu esimerkiksi automaattisten testien hylkäämisestä ohjelmiston toiminnallisuuden muututtua merkittävästi, väärien positiivisten esiintymisriskin kasvamisesta ja testaustyökalujen omistajuuden kokonaiskuluista [RaW06].

Automatisoidun testin yhteydessä löydetyistä virheistä 60-80 % löytyy osana testin kirjoittamisprosessia [Kan97]. Tämän jälkeen testien rooli muuttuu negatiivisesta positiiviseksi ja varmentavaksi [BWK05] eli ne varmentavat olemassa olevaa toimin-

nallisuutta ja tuottavat lisää tietoa ohjelmiston tilasta. Ketterien prosessimallien kannalta tämä on tärkeää, sillä ne korostavat tiedonkulun merkitystä projektissa ja ohjelmiston tilan pitämistä sellaisena, että se on millä hetkellä hyvänsä toimitettavissa asiakkaalle.

2.2.6 Synkronoitu, osittain synkronoitu ja synkronoimaton testaus

Synkronoitu testaus (in-sync testing) on iteraation sydämenlyöntien tahdissa tehtyä testausta, sydämenlyönti on yleensä työpäivän mittainen ajanjakso [Itk07]. Tästä ovat hyvänä esimerkkinä kehittäjien päivittäin kirjoittamat automatisoidut yksikkötestit. Testien valmistumista ja onnistunutta suorittamista seurataan sydämenlyönnin tasolla, vaikka niiden valmistuminen kestäisi useamman sydämenlyönnin ajan. Muita esimerkkejä synkronoidusta testauksesta ovat funktionaalisten järjestelmätestien suunnittelu ja toteutus, regressiotestien suoritus, testitulosten ja virheiden raportointi sekä virheenkorjausten varmistaminen. Synkronoitu testaus seuraa tiiviisti kehityksen sydämenlyöntien rytmiä, ja sen tilasta kommunikoidaan päivittäin.

Osittain synkronoitu testaus (off-sync testing) tapahtuu osana iteraatiota ja on merkitty osaksi sen laatutavoitteita. Esimerkkejä osittain synkronoidusta testauksesta ovat laajempien ohjelmistonosien automaattisen testauksen suunnittelu ja kirjoittaminen, raskaampien testausoperaatioiden suorittaminen, manuaalisen testauksen suorittaminen ja uusien testausmenetelmien kokeilu.

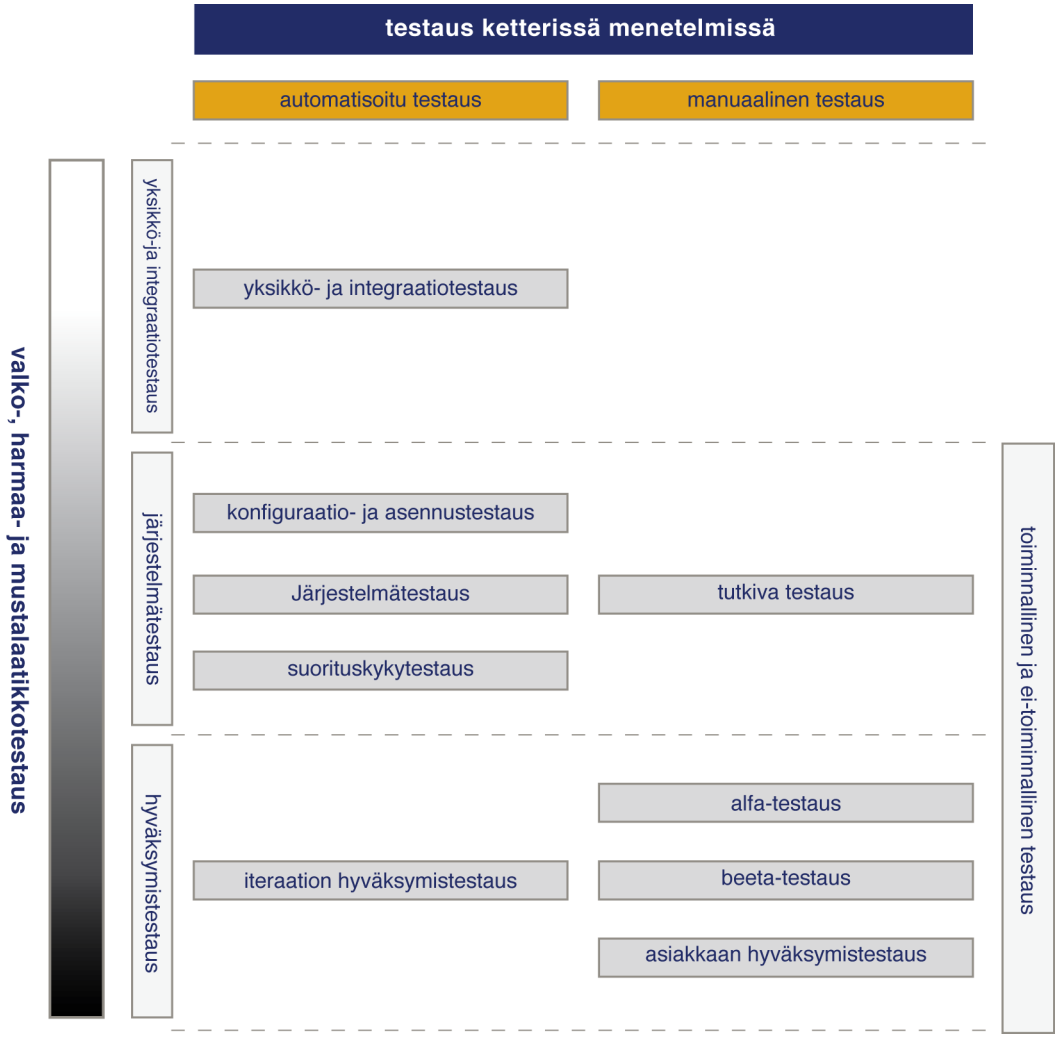
Synkronoidun testauksen ja osittain synkronoidun testauksen ulkopuolelle jää *synkronoimaton testaus (no-sync testing)*, jota ei ole sidottu mihinkään ajanjaksoon. Synkronoimattomassa testauksessa työn suunnittelu ja seuraaminen on vaikeaa. Parempi olisi liittää testaustoiminnot osaksi prosessin jonkin vaiheen tavoitteita.

2.2.7 Luokittelun rajat

Luokittelu on kankea tapa kuvata testausmenetelmien välisiä suhteita. Esimerkiksi iteraation hyväksymistestaus voidaan ymmärtää järjestelmätestauksena, joka toteutetaan tutkimusmatkailevana testauksena. Tällöin se olisi funktionaalista, manuaalista, positiivista ja osittain synkronoitua valkolaatikkotestausta.

Kuva 3 havainnollistaa testausmenetelmien sijoittumista luokkiin. Jakoa positiivisen ja negatiivisen testauksen välillä ei voida sijoittaa kuvaan, koska jokaista menetelmää voidaan käyttää kummallakin tavalla. Myös testien synkronointiaste vaihtelee: valkolaatikkotestit ovat todennäköisemmin synkronoituja ja harmaa- sekä mustalaa-

tikkotestit osittain synkronoituja. Musta-, harmaa- ja valkolaatikkotestausten sijoittumista kuvaa liukuma.



Kuva 3: Testausmenetelmät luokiteltuina eri perusteiden mukaan.

2.3 Yksikkö- ja integraatiotestaus

Yksikkötestaus (unit testing) kohdistuu pieneen osaan ohjelmistosta, kuten yksittäiseen metodiin, funktioon tai luokkaan [Bin00, s. 45]. Yksikkötestauksella varmistetaan yksittäisen osan sisäinen toiminta erilaisten ulkoisten syötteiden avulla. *Integraatiotestaus (integration testing)* kohdistuu luokkien, moduuleiden, osajärjestelmien ja järjestelmien väliseen toimintaan. Integraatiotestauksen tehtävä on varmistaa, että ohjelmiston eri osat toimivat yhdessä. Integraatiotestauksessa testataan erityisesti rajapintoja [Bin00, s. 45]. Integraatiotestausta joudutaan usein suoritta-

maan tilanteessa, jossa ohjelmiston moduuleista vain osa on valmiina. Tällöin puuttuville moduuleille kirjoitetaan tynkätoteutus (stub), joka simuloi moduulin toimintaa [Bei90, s. 545]. Ohjelmistokoodissa voi olla osia, joiden puhdas yksikkötestaus on mahdotonta. Testien kirjoittaminen vaatii monien olioiden ja muuttujien tiettyjen tilojen asettamista sekä useiden rajapintojen käyttämistä. Tämän johdosta yksikkö- ja integraatiotestauksen välinen ero on häilyvä.

Ketterissä prosessimalleissa painotetaan kehittäjän itsensä kirjoittamia automatisoituja yksikkö- ja integraatiotestejä, jotka parantavat ohjelmistokoodin laatua [Bec00, s. 9][CrH02, s. 245]. Test Driven Development (TDD) on hyvä esimerkki automatisoitujen yksikkö- ja integraatiotestien käytöstä kehitystyössä. TDD on kuitenkin enemmän suunnittelua parantava tekniikka kuin testausmenetelmä, sillä testiä kirjoittaessaan kehittäjä suunnittelee testattavan kohteen rakennetta. Tavoiteltavana on pidettävä tilaa, jossa testejä on paljon ja ne testaavat kattavasti ohjelmiston eri osia.

2.4 Järjestelmätestaus

Järjestelmätestaus (system level testing) kohdistuu koko järjestelmän toimintaan, ja sitä suoritetaan usein käyttöliittymän kautta [Bin00, s. 45]. Järjestelmätestauksella on Binderin mukaan kolme päätehtävää: löytää järjestelmätason virheitä, näyttää toteen, että testattava ohjelmisto toteuttaa vaatimuksensa ja selvittää, onko järjestelmä toimitettavissa asiakkaalle [Bin00, s. 718]. Perinteisesti järjestelmätestaus on manuaalista työtä, jota suunnitellaan ja suoritetaan omana vaiheenaan osana ohjelmistotuotantoprosessia. Ketterät prosessimallit ovat kuitenkin yksikkötestien hengessä lisänneet tarvetta automatisoiduille järjestelmätesteille [CrH02, s. 133-137]. Automatisoituun järjestelmätestaukseen soveltuvia työkaluja on ilmestynyt useita. Esimerkkinä on avoimen lähdekoodin Selenium [1], jonka käytöstä on saatu positiivisia kokemuksia [HoK06]. Selenium on testaustyökalu käyttöliittymältään www-pohjaisia ohjelmia varten. Selenium testaa ohjelmia käyttämällä yleisimpien www-selainten välityksellä. Järjestelmätestaus sisältää järjestelmän toiminnan eri näkökulmiin erikoistuneita testausmenetelmiä.

[1] Lisätietoja Seleniumista löytyy http://selenium.openqa.org/ [27.3.2008].

2.4.1 Konfiguraatio- ja asennustestaus

Konfiguraatio- ja asennustestauksessa kokeillaan ohjelmiston kykyä asentua ja toimia oikein kaikissa tuetuissa ympäristöissä käyttäen erilaisia konfiguraatioita [Bin00, s. 742-743][KBP02, s. 96]. Näiden ympäristötekijöiden yhdistelmänä syntyy suuri joukko erilaisia testattavia tiloja. Vähintään tulee testata jokainen erilainen laitteisto minimi- ja maksimiasetuksilla [Mye04, s. 138]. Testattavien tilojen valinnan apuna voidaan käyttää äärellisiä tilakoneita [Bei90, s. 363]. Lisäksi automatisoimalla voidaan testata suurempi joukko tiloja kuin manuaalisesti. Tämän tyyppisen testauksen hyödyt lisääntyvät, kun testaus liitetään osaksi muuta automatisoitua järjestelmätestausta. Konfiguraatio- ja asennustestauksella saadaan arvokasta tietoa ohjelmiston käyttäytymisestä erilaisissa tilanteissa.

2.4.2 Suorituskykytestaus

Suorituskykytestauksessa testataan järjestelmän kykyä suoriutua määritetyssä ajassa ja määritetyllä kuormalla ennalta määritetyistä tehtävistä [Bin00, s. 743-744]. Näin varmistetaan, että ohjelmisto täyttää asetetut suorituskykyvaatimukset. Testauksessa erotetaan ohjelmiston ulkopuolisten tekijöiden ja sisäisten osien vaikutukset suorituskykyyn. Ulkopuolisilla tekijöillä tarkoitetaan ohjelmistoa ympäröiviä toisia järjestelmiä ja fyysisiä laitteita. Sisäisiä osia ovat ohjelmiston eri moduulit ja osat.

Ketterään prosessiin sopiva suorituskykytestaus asettaa erityisvaatimuksia kehittäjille ja tekniikalle. Tiimin on oltava tehtävään motivoitunut ja vahvasti keskittynyt ratkaisemaan suorituskykyongelmia. Kehittäjillä on oltava teknistä kykyä sekä suorituskykytestaukseen että järjestelmän optimointiin. Tiimin käyttämän kehitysmenetelmän tulee olla iteratiivinen, ja tiimiin tulee kuulua suorituskykytestauksen asiantuntija. Tuotteen tulee asentua automaattisesti (nappia painamalla) ja suorituskykytestien tulee olla automatisoituja. [Dob07]

Onnistuneen testauksen tärkein edellytys ovat kuitenkin iteratiivisesti, laajassa yhteistyössä ja käyttäjien odotusten perusteella laaditut suorituskykyvaatimukset ja -testit. Vaatimukset antavat suorituskykytestaukselle kohteet ja tavoitearvot. Ketterässä prosessissa käyttäjien vaatimukset muuttuvat ajan kuluessa, jolloin on tärkeää, että myös suorituskykyvaatimukset pysyvät ajan tasalla. Täydellisten ja kattavien suorituskykyvaatimusten määrittäminen tuotantoprojektin alussa ei ole mielekästä, koska monimutkaisen, laajan ja muuttuvan tehtävän sisäistäminen vie aikansa.

Ratkaisuna on esitetty nelivaiheista suorituskykyvaatimusten laadintaa, jossa vaatimusten yksityiskohtaisuus kasvaa vaihe vaiheelta prosessin edetessä. Vaiheittaisen luonteensa johdosta malli soveltuu hyvin iteratiivisiin prosesseihin. Lisäksi malli helpottaa suorituskykytestien kirjoittamista osana kutakin vaihetta [HJW06]. Suorituskykytestit tulee ajaa osana jatkuvaa integraatiota, mikä ei aina ole teknisesti mahdollista [Pul06]. Testit voidaan ajaa myös iteraation lopussa ja huomioida testien tulokset seuraavassa iteraatiossa.

2.4.3 Tutkimusmatkaileva testaus

James Bach määrittelee *tutkimusmatkailevan testauksen (exploratory testing)* seuraavasti: "Tutkimusmatkaileva testaus on samanaikaisesti tapahtuvaa oppimista, testien suunnittelua ja testien suorittamista"[Bac07]. Tutkimusmatkaileva testaaja suorittaa testauksensa ilman tarkkaa testisuunnitelmaa, mutta usein kohdistaen sen johonkin johtoajatukseen, teemaan tai ohjelmiston osaan. Testauksen aikana käydään intuitiivisesti ja loogisesti läpi ohjelmiston eri toiminnallisuuksia ja osia. Välillä testaus poikkeaa mielenkiintoiseksi koetulle sivupolulle ja kokeilee ohjelmiston epäilyttäviä osia perusteellisemmin, mutta palaa aina lopuksi alkuperäiselle reitilleen. Testausta suoritetaan sekä manuaalisesti että automatisoituja testejä kirjoittaen. Suoritettavien testien yksityiskohdat riippuvat testaajan havainnoista testauksen edetessä.

Tutkimusmatkailevan testauksen tuloksena saadaan raportti, joka kertoo mitä on testattu, mistä löytyi virheitä tai mitä toiminnallisuuksia testaaja pitää epäilyttävinä ja jatkotestauksen arvoisina. Tutkimusmatkaileva testaus on luova ja opettava tutkimusmatka, jossa testaajan tulee käyttää tietojaan, taitojaan ja saatavilla olevia apuvälineitä löytääkseen mahdollisimman monta virhettä ohjelmistosta.

Tutkimusmatkailevan testauksen etuja ovat testaajien vapauttaminen joidenkin piirteiden tarkempaan ja syvempään testaukseen, hyvin ohjelmiston toiminnan tuntevien testaajien tehokkuus virheiden löytämisessä ja ohjelmiston tilan nopea selvittäminen [ItR05]. Testauksen toistettavuus on hankalaa, ja eri ohjelmistotoiminnallisuuksia ei testata kattavasti. Tutkimusmatkailevan testauksen tehokkuutta on vaikea verrata systemaattisempiin testausmuotoihin, koska asiaa ei ole tutkittu tarpeeksi. Uusi vertaileva tutkimus kuitenkin toteaa, että tutkimusmatkailevan testauksen ja suunniteltujen testitapausten kautta tehdyllä testauksella ei ole eroja löydettyjen virheiden määrissä tai tyypeissä, vakavien virheiden määrissä tai virheiden löytämisen vaikeudessa [IML07]. Toisaalta tutkimus osoitti, että tutkimusmatkaileva

testaus tuottaa vähemmän vääriä virheraportteja. Tutkimusmatkailevan testauksen ajattelumalli auttaa kehittäjiä kirjoittamaan parempia automatisoituja yksikkötestejä [Koh07a]. Tutkimusmatkaileva testaus ei vaadi suuria ennakkovalmisteluja ja sopii hyvin käytettäväksi nopeasyklisissä ketterissä ohjelmistoprosesseissa.

2.5 Hyväksymistestaus

Hyväksymistestauksella varmistetaan ohjelmiston olevan laadultaan ja ominaisuuksiltaan riittävän hyvä julkaisua varten [Bin00, s. 748-749][PeY07, s. 421-423]. Hyväksymistestausta voidaan kuvata *ohjelmistokehityksen jälkeen tulevana testauksena (Post-development Testing)*, eli sen edellytyksenä on aikaisempien testausvaiheiden, kuten yksikkö-, integraatio- ja järjestelmätestauksen, suorittaminen loppuun. Hyväksymistestaukseen on neljä erilaista näkökulmaa: iteraation hyväksymistestaus ketterien prosessimallien määrittelemällä tavalla, alfa-testaus ohjelmiston tuottaneen organisaation sisällä, beeta-testaus valikoidulla ulkoisella asiakasjoukolla ja asiakkaan itsensä suorittama hyväksymistestaus.

XP-prosessimallissa suositellaan käytettäväksi ainoastaan automatisoituja *iteraation hyväksymistestejä*, jotka kirjoitetaan osana jokaista iteraatiota [CrH02, s. 133-137]. Hyväksymistestit toimivat asiakkaan ja kehittäjien välisen kommunikaation välineenä, jolloin kumpikin osapuoli kykenee yksiselitteisesti määrittelemään iteraatiossa tuotettavien ohjelmistopiirteiden toiminnan ja laadun yksityiskohdat. Laadun yleisemmät linjat XP-prosessimallissa asetetaan osana suunnittelupeliä. Peräkkäisten iteraatioiden automatisoidut hyväksymistestit muodostavat kasvavan joukon testejä varmistamaan ohjelmiston toimintaa muutoksen alla. Ketterät menetelmät mahdollistavat alfa- ja beeta-testaukseen sopivien julkaisujen tekemisen jopa päivittäin. Silloin alfa- ja beeta-testaus olisivat paremmin synkroonissa ketterän tuotekehityksen kanssa. Hyväksymistestaus voidaan myös järjestään iteratiivisesti, jolloin asiakas maksaa jokaisesta kelvollisesta julkaisusta.

Alfa-testaus on ohjelmiston tuottaneen organisaation sisäinen hyväksymistestaus [Bin00, s. 748-749][PeY07, s. 421-423]. Tuote julkaistaan organisaation sisäiseen käyttöön, tai ohjelmistoa testaa itsenäinen ja kehityksestä riippumaton testausryhmä. Oikeaoppinen alfa-testaus tapahtuu laboratorio-olosuhteissa aidoilla käyttäjillä. Vaihtoehtoisesti todellisia käyttötilanteita voidaan simuloida mahdollisimman edustavalla käyttötapaus- tai käyttäjäjoukolla.

Beeta-testauksessa ohjelmisto julkaistaan organisaation ulkopuolelle joko valikoidul-

le asiakasjoukolle tai yleiseen jakeluun internetin kautta [Bin00, s. 748-749][PeY07, s. 421-423]. Asiakas tai yksittäiset ohjelmiston käyttäjät asentavat tuotteen omaan käyttö- tai testausympäristöön, kokeilevat ohjelmistoa tavallisimmilla käyttötavoilla ja raportoivat virheistä ohjelmiston tuottaneelle organisaatiolle. Tuottaja julkaisee beeta-version organisaation ulkopuolelle sillä varauksella, että ohjelmistosta voi löytyä vielä merkittäviä puutteita tai virheitä.

Asiakkaan suorittama hyväksymistestaus varmistaa tuotetun ohjelmiston vaatimusten mukaisen ja virheettömän toiminnan [Bei90, s. 16][Bin00, s. 748-749]. Testaus tapahtuu yleensä asiakkaan omissa tiloissa. Asiakkaan ja ohjelmistontuottajan tulee yhdessä sopia hyväksymistestauksen suunnitelmista ja järjestelyistä. Hyväksymistestauksen riittävä onnistuminen on usein ehtona sille, että asiakas maksaa tehdystä kehitystyöstä.

3 Ketterien prosessien testaus ja laadunvarmistus

Ketterissä menetelmissä käytössä olevat testaus- ja laadunvarmistustoiminnot eivät ole tarpeeksi kattavia, vaan niiden kehittämistä ja uusien testauskäytäntöjen lisäämistä prosesseihin tarvitaan [IRL05]. Tärkeintä on tunnistaa ketterien prosessien mahdollisuudet ja vaatimukset testaukselle, kuten aikajänteet ja tehostunut kommunikaatio [Rau04]. Testaus tulee sijoittaa luonnolliseksi osaksi prosessia. Se ei voi olla pelkästään erillinen toimenpide tai vaihe [Koh07b]. Testauksen automatisointi on ketterän testauksen avain [CrH02, s. 131-145][Pul06]. Automaation avulla pysytään mukana iteraatioissa ja saadaan jatkuvasti luotettavaa tietoa ohjelmiston tilasta.

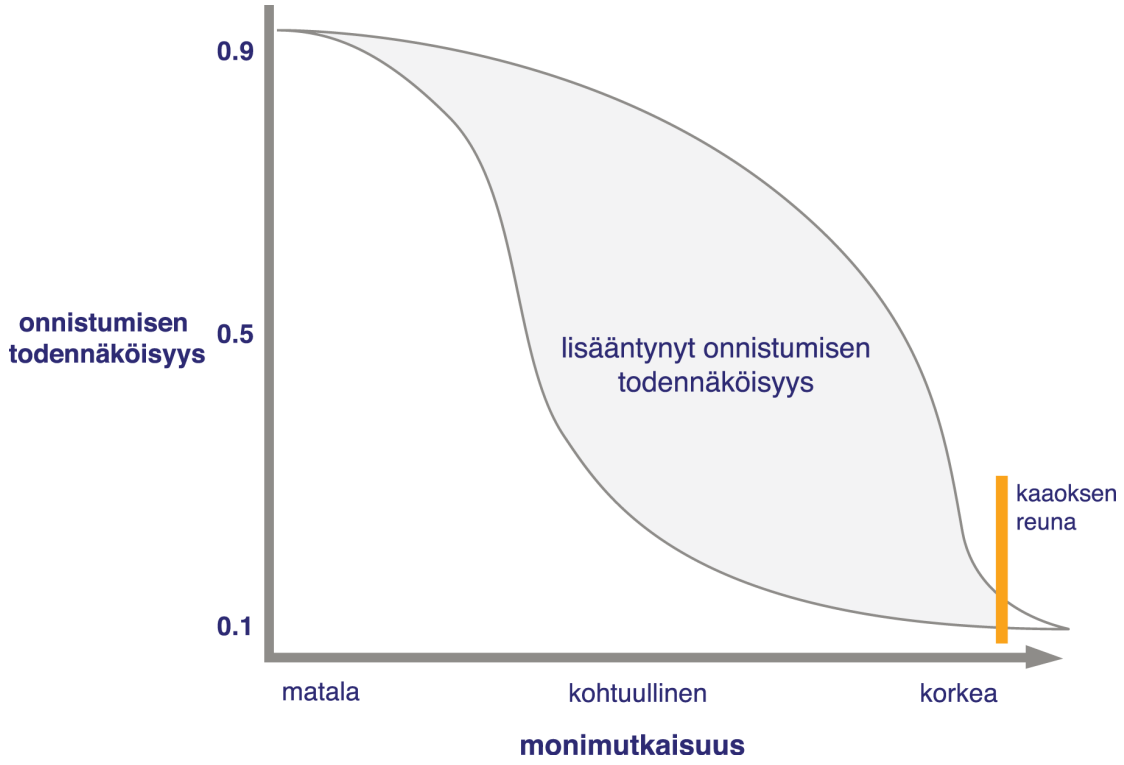
3.1 Ketterät prosessimallit

Modernin ohjelmistotuotantoprosessin hallittavuuteen vaikuttavat eniten ohjelmistolle asetetut vaatimukset, kehityksessä käytettävä teknologia ja kehitystyöhön osallistuvat ihmiset. Vaatimukset ovat monimutkaisia, moniselitteisiä, ristiriitaisia ja muuttuvia. Kehityksessä käytetty teknologia koostuu useista osista: ohjelmointikielistä, apukirjastoista, määrityksistä, protokollista ja kehitysympäristöistä. Teknologia on usein uutta, ainakin osittain kokeellista ja laadultaan vaihtelevaa. Eri teknologioiden yhdistelmä muodostaa helposti monimutkaisen kokonaisuuden. Ihmisten ei voida olettaa kehitystyössäkään toimivan koneen kaltaisesti, sillä he ovat tunteidensa ohjaamia ja heillä on vahvuuksia ja heikkouksia. [Sch04]

Ohjelmistoprosesseja voidaan vertailla sijoittamalla ne järjestyksen ja kaaoksen väliselle akselille, kuten kuvassa 4 on esitetty. Kaoottisuuden suuntaan prosesseja vievät vaatimusten, teknologian ja ihmismielen ongelmat. Prosesseja ympäröivät maailmat voidaan asettaa samalla akselille. Kaoottisissa maailmoissa ongelmat ovat monimutkaisia ja niiden ratkaisuun ei ole riittävästi resursseja. Järjestäytyneissä maailmoissa ongelmat ovat selkeitä ja hallittavia; niiden ratkaisuun on riittävästi resursseja. [Bac94]

Kaoottisuus ei ole ainoastaan negatiivinen ilmiö, vaan jopa eduksi ohjelmistoliiketoiminnalle jatkuvasti muuttuvassa markkina- ja kilpailutilanteessa. Ohjelmiston toiminnallisuuksien määrää voidaan lisätä kaksin- tai kolminkertaiseksi, jos samalla hyväksytään lisääntyvä hallitsemattomuus ja riski. Kaoottiset prosessit maksimoivat joustavuuden, hallitut prosessit ennustettavuuden. Kun ohjelmistoprojekti sallii lisääntyvän epävarmuuden optimoidakseen ohjelmiston sisällön, se samalla maksimoi

mahdollisuutensa menestymiseen. [Bac94]



Kuva 4: Ketterät menetelmät lisäävät onnistumisen todennäköisyyttä suhtautumalla joustavasti monimutkaisuuden ongelmiin [Sch96].

Ketterät prosessimallit on kehitetty toimimaan edellä kuvatussa maailmassa. Ketterien prosessimallien yleistymistä vauhditti vuonna 2001 tehty *'Ketterän ohjelmistokehityksen manifesti' (Agile Manifesto)*: "Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä itse ja auttamalla siinä muita. Tässä työssämme olemme päätyneet arvostamaan yksilöitä ja vuorovaikutusta enemmän kuin prosesseja ja työkaluja, toimivaa sovellusta enemmän kuin kokonaisvaltaista dokumentaatiota, yhteistyötä asiakkaan kanssa enemmän kuin sopimusneuvotteluita, muutokseen reagoimista enemmän kuin suunnitelman noudattamista. Vaikka jälkimmäisillä on arvoa, me arvostamme edellisiä enemmän."[Bec01]. Ketterät prosessimallit ovat ennen kaikkea ihmiskeskeisiä ja niiden tehokkuus perustuu joustavuuteen muuttuvien ja monimutkaisten olosuhteiden edessä [Sch96, HiC01].

Ketterät menetelmät ovat nykyisin hyvin suosittuja. Termi "ketterä"liitetään liian useaan asiaan ja alkaa menettää siten merkitystään. Termiin suhtaudutaan tunteella, eikä ketterien menetelmien tehoa tarvitse todistaa. Ketteryys on hyvää ja toimivaa, koska ketteryys on hyvää ja toimivaa. Jälki-agilismi *(post-agilism)* kritisoi

tätä sokeaa suhtautumista ketteryyden kaikkivoipaisuuteen. Kritiikin mukaan ketterissä menetelmissäkin on puutteita eivätkä ne ratkaise kaikkia ongelmia. Ketterien menetelmien käytön tulee olla perusteltua, ja niiden soveltamisessa pitää käyttää tilannekohtaista harkintaa. Ketterien menetelmien kehittyminen ja seuraavan prosessimallin syntyminen edellyttää kritiikkiä nykyistä kohtaan. Ketterät menetelmät eivät ole hopealuoti kaikkien ongelmien maagiseen ratkaisemiseen. [Koh06, Gor06, Mar03, Koh06, Gor07]

Vaikka tutkimus keskittyy testaukseen nimenomaan Scrum-prosessimallissa, on hyödyllistä esitellä myös Extreme Programming (XP) -prosessimalli. Scrum on ainoastaan prosessimallin aihio - se ei määrittele yksityiskohtaisia kehitysmenetelmiä. XP taas on varhaisemmin yleistynyt, täydellisempi ja monelta osin käytännönläheisempi prosessimalli. XP:n kehitysmenetelmiä käytetään usein osana Scrumin toteutusta täydentämään sen puutteita. Scrum ei esimerkiksi määrittele ohjelmointiteknisiä käytäntöjä (engineering practices), kuten pariohjelmointia tai refaktorointia. Muita ketteriä prosessimalleja ovat Adaptive Software Development (ASD), Crystal, Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD), Lean Development ja Rational Unified Process (RUP) [ASR02].

3.1.1 Extreme Programming (XP)

Extreme programming (XP) -prosessimallin ytimenä on tiimi, jossa kehittäjien lisäksi ovat liiketoiminnan sekä asiakkaan tai asiakasympäristön tarkasti tuntevat edustajat [Bec00, s. 139-148]. Asiakkaan apuna voi olla testaajia määrittelemässä hyväksymistestejä ja analyytikkoja määrittelemässä ohjelmiston tarkkoja vaatimuksia. Näiden roolien lisäksi tiimiin voi kuulua prosessin resursoinnista vastaava manageri ja osallistujia prosessimallin oikeassa käytössä tukeva valmentaja. Roolit eivät ole toisiaan poissulkevia ja ovat vapaasti yhdisteltävissä samoille ihmisille. Parhaimmillaan tiimissä ei ole yksittäisiä asiantuntijoita, vaan jäsenet taidoillaan täydentävät toisiaan.

XP-prosessimalliin kuuluu kaksiosainen suunnittelupeli, jolla määritetään ohjelmiston yleiset ominaisuudet ja kunkin iteraation työtehtävät [Bec00, s. 85-96]. Julkaisun suunnittelussa asiakas määrittää ohjelmistolta haluamansa piirteet ja tiimi arvioi kunkin ominaisuuden toteuttamiseen tarvittavan työmäärän. Arvion perusteella asiakas laatii summittaisen projektisuunnitelman. Arviot ja suunnitelmat ovat tässä vaiheessa tarkoituksella epätarkkoja, ja prioriteettien keskinäinen järjestys voi olla avoin. Projektisuunnitelma tulee jatkossa tarkentumaan osana jokaista iteraatiota. Iteraation suunnittelussa asiakas valitsee tiimin työtehtävät pariviikkoisen iteraatio-

jakson ajaksi. Tiimi määrittää asiakkaan toiveet tarkemmiksi työtehtäviksi ja sitoutuu sopivaksi arvioimaansa työmäärään. Tämän menetelmän avulla asiakas ohjaa projektia tietoisena kehitystyön tilasta ja etenemisestä.

XP käyttää jatkuvaa integraatiota eli menetelmää, jossa kehittäjät jatkuvasti tallettavat koodiaan yhteiseen versionhallintajärjestelmään. Jokaisen päivityksen jälkeen järjestelmä automaattisesti rakentaa tuotteen, ajaa sille määritellyt testit ja raportoi onnistumisesta ja mahdollisista virheistä. Nopeasti löydettyjen virheiden korjaus on helppoa ja tehokasta. Samalla on hyvä automatisoida asennus mahdollisimman realistiseen tuotantoympäristöön ja ajaa järjestelmätason testejä. Käytännössä tämä tarkoittaa sitä, että kehittäjät pysyvät jatkuvasti tietoisina tuotteen tilasta. Näin syyt löydettyihin ongelmiin ovat helposti ja nopeasti tunnistettavissa ja korjattavissa.

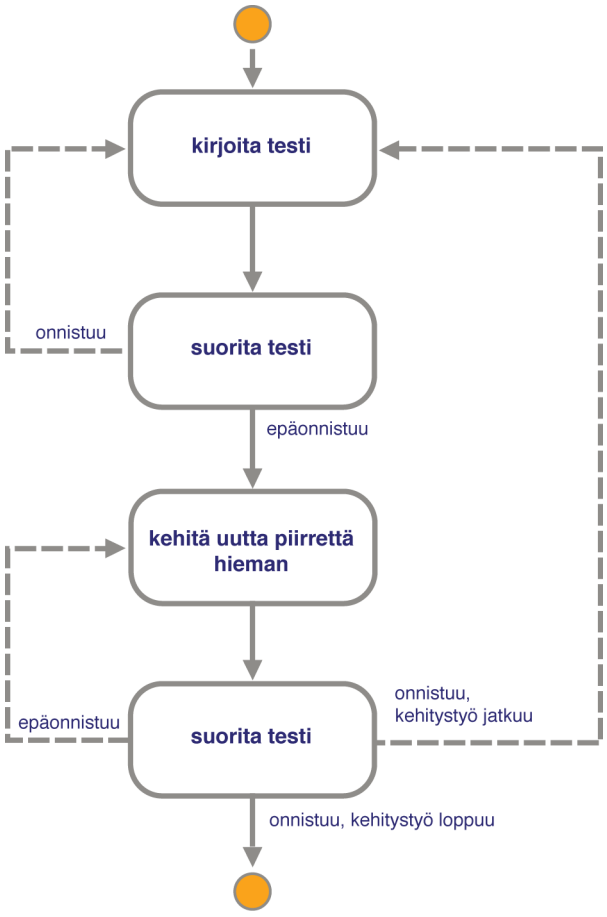
Jatkuva integraatio pienentää laadun riskiä ja lisää läpinäkyvyyttä [Fow01]. Jatkuvan integraation avulla tuote pidetään jatkuvasti asiakkaalle toimitettavassa tilassa, mikä edistää tuotteen laatua ja varmistaa oikeaa sisältöä [Bec00, s. 56]. Jokaisen iteraation jälkeen tuote sisältää asiakkaan haluamat uudet piirteet ja on täysin toimiva. Asiakas voi käyttää tuotetta sisäiseen arviointiin tai mieluiten julkaista sen loppukäyttäjille. Tuotteen ja sen tilan julkisuus edistää avointa tiedonkulkua, mikä taas osaltaan parantaa tuotteen laatua.

Asiakas määrittelee jokaiselle ominaisuudelle hyväksymistestit, jotka varmistavat sen oikeellisuuden [Bec00, s. 143-144]. Nämä testit tulee saman tien automatisoida, jotta kehityskiire ei estä niiden säännöllistä suorittamista. Pysyvänä osana jatkuvaa integraatiota testit auttavat varmistamaan tuotteen toiminnallisuuksia jatkossa.

Tuotteen arkkitehtuuri pidetään mahdollisimman yksinkertaisena; rakenteen tulee mahdollistaa vain olemassa oleva toiminnallisuus [Bec00, s. 57]. Ei kannata haaskata turhia resursseja suunnitteluun, jota ei välttämättä tulla tarvitsemaan. Toisaalta rakenteen pitää sallia jatkuvat lisäykset ja muutokset. Parannuksia ohjelmiston rakenteeseen syntyy osana iteraatioiden ja julkaisujen suunnittelua. Rakennetta parannetaan edelleen refaktoroinnin tai pienten ad hoc -suunnittelupalaverien avulla.

Kaikki ohjelmistokoodi kirjoitetaan pariohjelmointina käyttäen Test Driven Development -tekniikkaa (TDD). Pariohjelmoinnissa kaksi ohjelmoijaa istuu saman koneen ääressä; toinen kirjoittaa ja toinen avustaa ja kommentoi [Bec00, s. 53-62]. Näin kaikki tuotettu koodi tulee arvioitua ainakin kahden henkilön toimesta. Pariohjelmointi edistää tiedonkulkua ja tuottaa laadukkaampaa, paremmin suunniteltua ja testattua koodia kuin yksin ohjelmointi. Lisäksi pariohjelmointi kokeneemman oh-

jelmoijan kanssa on hyvä tapa oppia ennalta tuntemattoman koodin toimintaa ja kehittyä ohjelmoijana. TDD:ssa uudet piirteet kehitetään kirjoittamalla niille ensin testejä. Käytännössä kehittäjä kirjoittaa ensin yksinkertaisen testin ja vasta tämän jälkeen ohjelmoi piirrettä sen verran, että testi menee läpi [Bec00, s. 9]. Seuraavaksi hän parantaa vuorotellen testiä ja koodia ja jatkaa sykliä, kunnes on tyytyväinen ohjelmoituun piirteeseen ja testeihin. Täydentyneet testit lisätään yhteiseen testikantaan ja suoritetaan aina ohjelmiston muuttuessa. TDD:n käyttö vähentää selvästi virheiden määrää koodissa, helpottaa virheiden korjausta, selkiyttää uusia ohjelmistopiirteitä ja tuottaa laadukkaampaa koodia [Cri06b]. Aloitettaessa TDD:n käyttöä tiimin tuottavuus tulee hieman laskemaan, koska uuden tekniikan omaksuminen vie aikaa. TDD:n testien laatua voidaan parantaa tutkimusmatkailevan testauksen avulla, sillä monesti tämä menetelmä löytää kehitysvaiheessa helposti ratkaistavia virheitä [Koh07a]. TDD on kuitenkin enemmän suunnittelua kuin testausta tukeva käytäntö, sillä testiä kirjoittaessaan kehittäjä suunnittelee testattavan kohteen rakennetta.



Kuva 5: Esimerkki TDD:sta uml-kaaviona [Bec00, s. 9].

XP:ssa käytetään refaktorointia, jossa uudelleen kirjoittamalla parannetaan ohjelmiston rakennetta muuttamatta ulospäin näkyvää toiminnallisuutta [Bec00, s. 58]. Refaktoroinnilla on tarkoitus selkeyttää koodia jotta uusien toiminnallisuuksien lisääminen olisi helpompaa [Fow99]. Kattavat automatisoidut testit ovat refaktoroinnin edellytys ja sivutuote.

Refaktorointi ei ainoastaan helpota uusien piirteiden lisäämistä, vaan auttaa arvioimaan ja parantamaan samalla koko osa-alueen koodia. Erityisesti kannattaa refaktoroida toistettu koodi sekä sellaiset moduulit, joissa on heikko sisäinen yhtenäisyys tai liian tiivis kytkentä muihin moduuleihin. Moduulin tai funktion korkea sisäinen yhtenäisyys tarkoittaa, että se tuottaa yhden selkeästi määritellyn (ja ainutlaatuisen) toimenpiteen. Jos moduulissa on useita toisistaan riippumattomia toimintoja, yhden toiminnallisuuden muuttaminen lisää muiden virheistymistä. Matalassa kytkennässä eri moduulien välillä on vain yksi tai muutama yhteys. Tällöin yksittäistä moduulia voidaan muuttaa kasvattamatta riskiä jonkin toisen moduulin toiminnan rikkoutumiseen. Moduulien matala kytkentäaste ja korkea sisäinen yhtenäisyys lisäävät ohjelmiston stabiiliutta [SMC74, Mye75]. Yleisimpiä refaktorointitekniikoita ovat metodien, muuttujien ja luokkien uudelleennimeäminen paremmin tarkoitustaan kuvaaviksi ja metodien pilkkominen pienempiin ja paremmin ymmärrettäviin osiin. Yleisesti käytetään myös tiedon piilotusta, jolloin tiedon sisältävä kenttä ei näy luokasta ulospäin, mutta on käytettävissä liittymien kautta. Kaiken kaikkiaan refaktorointitekniikoita on olemassa yli 70 kappaletta [Fow99].

Kaikki XP:ssa tuotettu koodi on yhteisessä omistajuudessa eli ohjelmoijaparien vapaasti muokattavissa [Bec00, s. 59]. Yhteisomistajuus yhdistettynä refaktorointiin vähentää toistoa ja parantaa koodin laatua. Uusien ohjelmistopiirteiden lisääminen on nopeampaa kuin ilman yhteisomistajuutta, sillä kaikki voivat tarvittaessa keskittyä samaan moduuliin. Yhteisomistusta helpottavat koodausstandardien noudattaminen, refaktorointi ja TDD.

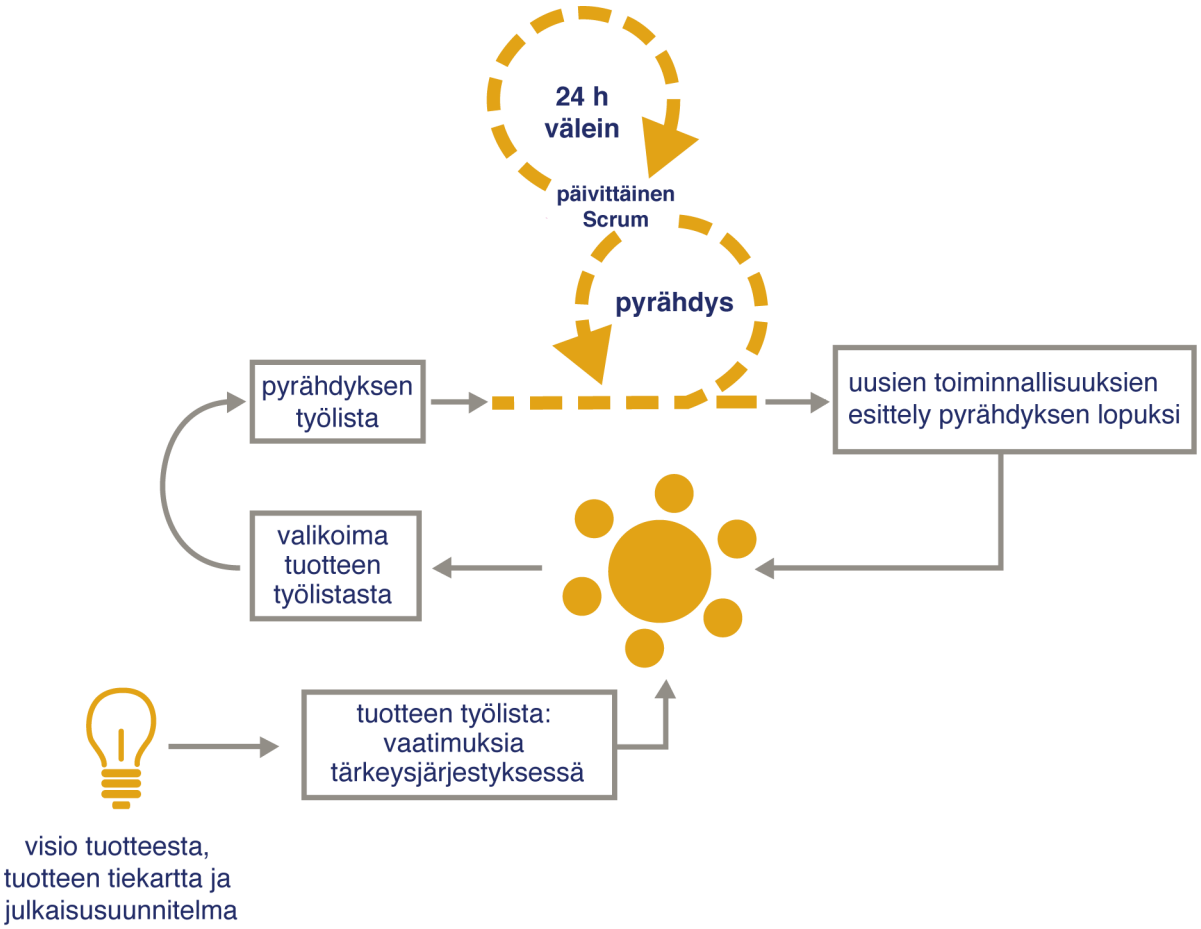
XP-tiimi luo yhteisen mielikuvan tai ajatusmallin rakennettavan ohjelmiston toiminnasta [Bec00, s. 56]. Tämä mielikuva auttaa kommunikoimaan kehitystyön aikana ohjelmistosta, sen toiminnasta ja rakenteesta. Tiimi työskentelee projektissa pitkällä tähtäimellä järkevällä työtahdilla. Ylitöitä tehdään ainoastaan väliaikaisesti, sillä jatkuvat ylityöt heikentävät käytäntöjä ja laatua puhumattakaan tekijöiden henkisestä hyvinvoinnista.

3.1.2 Scrum

Scrumissa [TaN86, DeS00] on kolme pääroolia: tuotteen omistaja (*product owner*), tiimi ja Scrum-mestari [Sch04, s. 1-14]. Tuotteen omistajalla on vastuullaan projektin rahoitus, liiketoiminnallinen kannattavuus (*return on investement, ROI*) ja tuotettavien ominaisuuksien tärkeysjärjestys. Tiimi on itseohjautuva ja itseorganisoituva. Tiimin vastuulla on teknologia sekä ratkaisut ohjelmiston ominaisuuksien tuottamiseen. Scrum-mestari vastaa Scrum-prosessista, sen opettamisesta projektiin osallistuville, sen mukauttamisesta organisaatioon ja Scrumin mukaisesta toiminnasta projektissa. Tuotteen omistaja, tiimi ja Scrum-mestari sitoutuvat prosessiin, yrityksen muut työntekijät ainoastaan osallistuvat siihen.

Scrumin keskeisin osa on pyrähdys *(sprint)* [Sch04, s. 1-14], joka on esitetty kuvassa 6. Pyrähdys kestää yleensä (neljä viikkoa eli) noin 30 päivää, mutta kesto saattaa vaihdella. Sprintin alussa tiimi selvittää, mitä sen kestäessä on tarkoitus saada aikaan. Tiimi valitsee yhdessä tuotteen omistajan kanssa ne toiminnallisuudet, jotka sen mielestä on mahdollista toteuttaa, ja sitoutuu tähän tavoitteeseen. Tämän jälkeen tiimi suorittaa kehitystyönsä itsenäisesti. Tiimi kokoontuu jokaisena työpäivänä Scrum-palaveriin, jossa jokainen kertoo edistymisensä edellisen palaverin jälkeen, mitä aikoo tehdä ennen seuraavaa palaveria ja mitä esteitä on onnistumisen tiellä. Palaverien ja muun kommunikaation avulla tiimi tekee itsenäisesti tarvittavat tarkennukset työskentelyynsä. Sprintin lopussa tiimi esittelee iteraation tuotokset asianosaisille. Tällöin uudet toiminnallisuudet varmennetaan ja koko projektin aikataulua sekä suunnittelua voidaan tarkistaa. Sprintin lopussa tiimi pitää retrospektiivin eli arviointipalaverin, jossa jokainen tiimin jäsen antaa palautetta ja kehitysehdotuksia seuraavaa pyrähdystä varten.

Tuotteen työlista (product backlog) on priorisoitu lista tuotteeseen halutuista toiminnallisuuksista [Sch04, s. 1-14]. Tuotteen omistaja käyttää työlistaa huolehtiakseen, että oleellisimmat osat tuotteesta rakennetaan ensin. Listassa jokaiselle toiminnallisuudelle on arvioitu prioriteetti ja toteuttamiseen tarvittava työmäärä. Lista muuttuu koko ajan liiketoiminnallisten vaatimusten ja teknologian mukaan. *Sprintin työlista (sprint backlog)* on pyrähdyksen alussa tehtävä priorisoitu lista, jossa on määritelty pyrähdyksen aikana tehtävät toiminnallisuudet [Sch04, s. 1-14]. Listaa ylläpidetään läpi pyrähdyksen. Listan toiminnallisuudet jaetaan pienempiin osiin eli tehtäviin. Tehtävistä tehdään yleensä alle kuuden tunnin mittaisia. Sprintin työlistasta näkyy jokaiseen tehtävän tila (aloittamatta, työn alla tai tehty), suorittamiseen vielä kuluva aika sekä vastuussa olevat tiimin jäsenet.

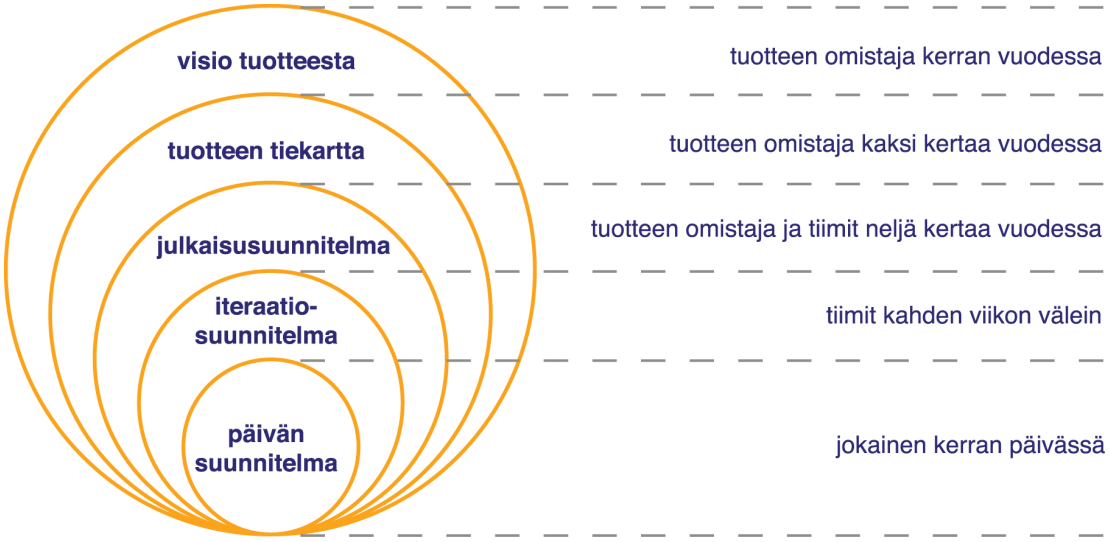


Kuva 6: Scrumin iteraatio [Sch04, s. 9].

Tuotteen tulee olla toimitettavissa asiakkaalle uusine toiminnallisuuksineen jokaisen pyrähdyksen jälkeen [Sch04, s. 1-14]. Toiminnallisuus on valmis vasta, kun ohjelmiston rakenne on hyvä, sen tuottava koodi on selkeää ja testattu, ohjelmistosta on tehty käynnistyvä versio ja ohjeet uusien ominaisuuksien käyttämiseksi ovat valmiit.

Erityisesti laajempien ohjelmien tuotannossa ja käytettäessä useita Scrum-tiimejä on tärkeää huomioida tuotteen työlistan yläpuolella olevat kolme suunnittelun tasoa: *visio tuotteesta (product vision)*, *tuotteen tiekartta (product roadmap)* ja *julkaisusuunnitelma (release plan)*. Tasot on esitetty kuvassa 7. Tuotteen vision laatii tai tarkistaa vuosittain tuotteen omistaja. Visiossa kuvataan tuotteen haluttu tila projektin päättyessä ja osoitetaan priorisoidussa järjestyksessä, minkä osien tuotteessa tulee muuttua ja mitä resursseja tähän on käytettävissä. Vision pohjalta tuotteen omistaja laatii kaksi kertaa vuodessa tuotteelle tiekartan, jossa kuvataan puolivuosittaisissa jaksoissa tuotteen julkaisut ja niiden toiminnallisuudet. Julkaisusuunnitelman laativat tiekartan pohjalta tuotteen omistaja ja tiimi neljännesvuosittain.

Julkaisusuunnitelmassa kuvataan julkaisujen ajankohdat, teemat ja suunnitellut toiminnallisuudet. Käytettäessä useita tiimejä julkaisut jaetaan tiimikohtaisesti, joten tiimien iteraatioiden on oltava samassa rytmissä. [Smi07]

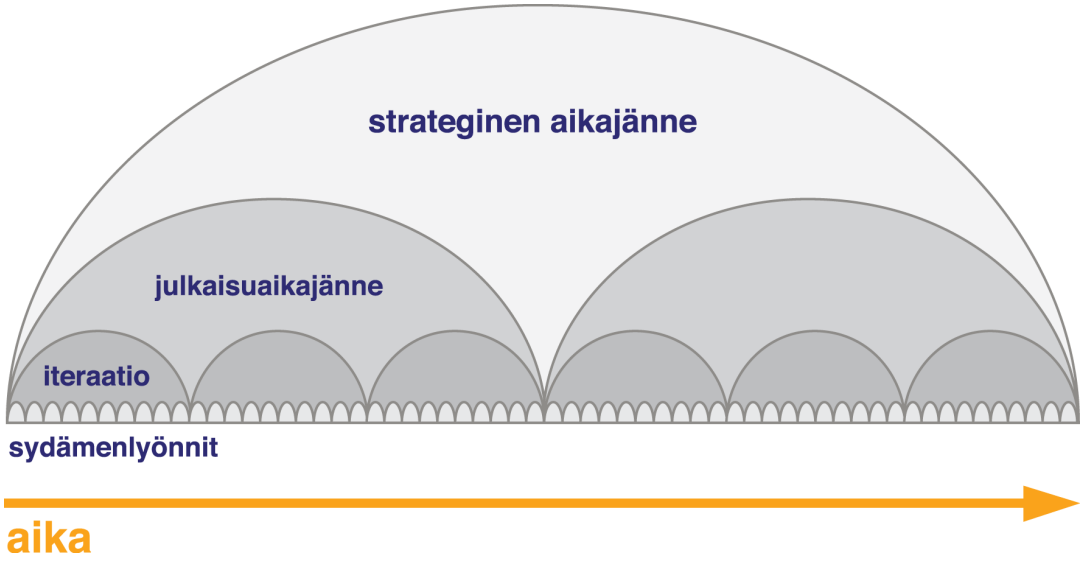


Kuva 7: Suunnittelun viisi tasoa ketterässä ohjelmistoprosessissa [Smi07].

3.2 Testausmenetelmät aikajänteillä

Ketterässä prosessimallissa tapahtuvaa tuotekehitystä voidaan kuvan 8 havainnollistamalla tavalla mallintaa jakamalla se neljään aikajänteeseen: sydämenlyöntiin, iteraatioon, julkaisuun ja strategiaan [Rau04]. Aikajänteet ovat ajallisesti päällekkäisiä abstraktiotasoja. Ne auttavat löytämään oikean aikavälin toimenpiteiden suorittamiseen tai tarkasteluun.

Ketterissä menetelmissä tiimi on vastuussa tuotteen laadusta ja sitä kautta kaikista tarvittavista testausmenetelmistä. Scrumissa tiimi neuvottelee tuotteen omistajan kanssa tuotteen laatuun liittyvät toimenpiteet osaksi iteraatioita. Tuotteen laadusta vastaa tuotteen omistaja ja tiimi on vastuussa laatutarpeiden esille tuomisesta ja perustelemisesta. Varsinkin strategisella aikajänteellä tapahtuva laadun kehittäminen on tuotteen omistajan tai hänen yläpuolellaan organisaatiossa olevien vastuulla ja resursoitavissa.



Kuva 8: Ketterän prosessimallin aikajänteet [Rau04].

taso	vastuullinen	aika	laatutoimenpide
tuotteen visio	tuotteen omistaja	1 vuosi	laatusuunnitelma
tuotteen tiekartta	tuotteen omistaja	1/2 vuosi	testausstrategia
julkaisusuunnitelma	tuotteen omistaja+tiimi	1/4 vuosi	hyväksymistestaus
iteraatiosuunnitelma	tiimi	2 viikkoa	ATDD+muu testaus
päivän suunnitelma	yksilöt	1 päivä	TDD

Taulukko 1: Scrum-suunnittelun tasot, vastuut ja aikajaksot sekä vastaava laatudokumentti/toimenpide

3.2.1 Sydämenlyönti – tiimi testaa

Sydämenlyönti (heartbeat) on ajanjakso tunneista päiviin tai viikkoihin ja rajoittuu tehtäviin, joissa kehitystä verrataan laadittuihin suunnitelmiin [Rau04]. Scrum-prosessimallissa tämä sydämenlyönti sijoittuu päivittäisten Scrum-palaverien välille. Sydämenlyöntiin sisältyy useita integraatioita: ohjelmiston rakentamista ja kaikkien ohjelmistolle kirjoitettujen automatisoitujen testien suorittamista. Usein integraatio on jatkuvaa. Testien tulokset kuvaavat ohjelmiston tilaa ja refaktoroinnista ja muutoksista aiheutuneet virheet löytyvät nopeasti. Testien tulee käyttää monipuolisesti erilaisia testausmenetelmiä. Testit tulee ajaa iteraation aikana, jolloin huomiot ja virheet välittyvät suoraan prosessiin. Iteraation jälkeen suoritetut testit aiheuttavat lisätyötä ja viivästymistä. Palautesyklin pituus tulee minimoida tehokkuuden lisäämiseksi.

Yksikkötestien kirjoittaminen on tärkein sydämenlyönnin aikana tehtävistä laadunparannustoimenpiteistä. Ne voidaan kirjoittaa ennen ohjelmistokoodia (TDD) tai välittömästi sen jälkeen [Els07, s. 67-77]. Testien kirjoittaminen ennen ohjelmointia on työläämpää, mutta silloin testit tulevat varmemmin kirjoitetuksi. Etukäteen kirjoitetut testit vaikuttavat ohjelmointitehtävän ymmärtämiseen ja suunnitteluun. Jälkikäteen kirjoitetut testit ovat puolueellisia, sillä useimmat kehittäjät pyrkivät samalla alitajuisesti vahvistamaan käsitystään oikeasta tavasta käyttää ohjelmoitua piirrettä eivätkä löytämään virheitä ajattelustaan. Tähän perustuu Weinbergin laki, jonka mukaan kehittäjän ei tule yksin testata omaa koodiaan [Wei71]. Kehittäjät suorittavat kirjoittamansa testit ensin paikallisesti ja sen jälkeen osana jatkuvaa integraatiota. Ruby-ohjelmointikielelle on tarjolla TDD:ta nopeuttava Autotest-kirjasto, joka käy jatkuvasti läpi projektin tiedostoja ja havaitessaan muutoksia ajaa osa-alueen testit. Näitä testejä ajetaan, kunnes ne menevät läpi. Tämän jälkeen vielä suoritetaan kaikki muut testit. Autotestin avulla kehittäminen nopeutuu, koska testejä ei tarvitse itse käynnistää, vaan ne toimivat koko ajan aktiivisesti. Vaikka ohjelmoijan kirjoittamat yksikkötestit vahvistavatkin regressiotestausta, ne palvelevat enemmän ohjelmiston rakenteen kehittämistä kuin testausta. Tämän vuoksi tarvitaan myös muunlaista testausta.

Behaviour Driven Development (BDD) on kehittyneempi versio TDD:sta. BDD:n perusajatuksena on määritellä ohjelmistokoodin toiminta ja liiketoimintalogiikka ensin luonnollisella kielellä ja vasta sitten toteuttaa se ohjelmoimalla. Samalla koodi tulee suunniteltua ennen sen kirjoittamista. BDD:ssa kirjoitetaan ensin kuvaus ajatuksesta, josta saadaan johdettua määritys, josta vuorostaan saadaan automatisoitu testi. Luonnollisella kielellä tehty kuvaus auttaa liiketoimintaa ymmärtämään ohjelmistokoodin toimintaa. Siten kehittäjät helpommin keskittyvät asiakkaan kannalta tärkeisiin toiminnallisuuksiin eivätkä juutu teknisiin yksityiskohtiin. Luonnollinen kieli on myös paljon ilmaisuvoimaisempi ja kehittäjille intuitiivisesti loogisempi kuin TDD-kirjastoissa käytetyt totuusarvoiset assertiot. BDD:ta varten on olemassa ainakin kaksi avoimen lähdekoodin kirjastoa eli Javalle JBehave ja Rubylle RBehave. [Ast05]

3.2.2 Iteraatio

Hyväksymistestaus on oleellinen osa iteraation aikaista laadunvarmistusta. Uusien ohjelmapiirteiden läpimenneet hyväksymistestit siirtävät vastuun niiden laadusta ja oikeellisuudesta tiimiltä tuotteen omistajalle. Hyväksymistestaus kannattaa suorit-

Vaihe	**toimenpide**
1)	Valitse tärkein ominaisuus, joka järjestelmässä on toteutettava.
2)	Kirjoita tälle ominaisuudelle user story.
3)	Toteuta storyn ensimmäinen askel => FAIL.
4)	Kirjoita askeleen vaatima view spec => FAIL.
5)	Toteuta näkymään askeleen vaatima toiminnallisuus => view spec menee läpi, storyn steppi (ehkä) edelleen FAIL.
5b)	Refaktoroi näkymää tarpeen mukaan. Tämä saattaa vaatia toiminnallisuuden siirtämistä helpereihin (ja niiden speksaamista).
6)	Kirjoita askeleen vaatimalle controller-toiminnolle speksi. Rajapinta toiminnolle saadaan siitä, mitä näkymän speksissä on jouduttu mockaamaan.
7)	Toteuta toiminnallisuus controlleriin niin, että askeleen vaatimat controller-speksit menevät läpi.
7b)	Refaktoroi
8)	Kirjoita askeleessa käytettäville ActiveRecord-luokille speksit, joiden avulla ne toteuttavat kohdissa 4 ja 6 mockatut rajapinnat.
9)	Toteuta edellisen kohdan speksit toteuttava toiminnallisuus.
9b)	Refaktoroi tarpeen mukaan.
10)	Palaa kohtaan kolme ja seuraavaan askeleeseen.

Taulukko 2: Jarkko Laineen kuvaama vaiheittainen toteutus on hyvä esimerkki BDD:sta. Siinä esitetään vaiheittain Model-View-Controller-arkkitehtuurissa tarvittavien komponenttien määrittely ja kirjoitus Ruby on Rails -kirjaston avulla. Toteutuksessa kirjoitetaan ensin integraatiotesti ja sitten testit jokaisella komponentille. Ulomman kerroksen määrittely testiolioiden avulla auttaa alemman kerroksen luokkien rajapintojen kehittämisessä. Esimerkki löytyy Ruby on Rails: Finnish -keskusteluryhmästä otsikolla Story runner ja rspec käytännössä, 13.2.2008, http://groups.google.com/group/finnishrails/browse_thread/thread/17d3922e09b3edeb [10.3.2008]

taa iteraation aikana, vaikka tuote ei vielä olisi ehjä ja helposti testattavissa. Yhteistyö kehittäjien kanssa on kuitenkin tällöin tehokkainta.

Hyväksymistestauslähtöisessä kehityksessä (acceptance test-driven development, ATDD) hyväksymistestit kirjoitetaan iteraation alussa [Pul06, Ran07]. Testien kirjoittamisen voivat hoitaa yksin tai yhteistyössä testaaja, kehittäjät tai asiakas. Testien kirjoittaminen selkiyttää tiimin tavoitteita. Liian yksityiskohtaisten testien kirjoittaminen ja määrittely voi hämmentää kehittäjiä. Tehty kehitystyö voi johtaa testien työlääseen uudelleenkirjoittamiseen. Ongelmat voidaan välttää kirjoittamalla yleisluontoiset hyväksymistestit ennen iteraation alkua ja yksityiskohtaiset hyväksymistestit samanaikaisesti tuotantokoodin kirjoittamisen kanssa. Tämä lähestymistapa edellyttää tarkkaa viestintää toimiakseen.

Hyväksymistestien laatiminen sopivalla työkalulla helpottaa toiminnallisten vaatimusten määrittelyä, kommunikointia ja validointia [RMM05, MMC06]. Esimerkkinä tällaisesta ovat *avainsanaohjatun testiautomaation (keyword-driven test automation)* työkalut, kuten Fitnesse [2]. Keskivertoasiakkaalla voi kuitenkin olla hankaluuksia vaatimusten tarkassa määrittelyssä liian teknisen työkalun kautta. Liiketoimintalogiikan ja käyttöliittymän liian tiukka kytkentä voi estää iteraation aikaisten automaattisten testien kirjoittamisen, jolloin ohjelmiston arkkitehtuuria tulee muuttaa [Sum07].

Iteraation aikana tuotteen ominaisuuksien kehittymistä seurataan asennus-, konfiguraatio- ja suorituskykytestauksen avulla. Monet sydämenlyönnin laadunvarmistusta tukevat toimenpiteet tehdään iteraation aikana [IRL05]. Tällaisia ovat esimerkiksi yksikkötestien lisäksi muiden automatisoitujen testien kirjoittaminen ja suunnittelu. Muitakin laadunvarmistuksen tehtäviä voidaan sisällyttää osaksi iteraation tavoitteita, jolloin niiden seuranta ja resursointi tulee hoidetuksi yhdessä tiimin kanssa.

3.2.3 Julkaisuaikajänne

Jos testausta tai virheidenkorjausta ei sisälly iteraatioihin tarpeeksi tasapainottamaan uusia toiminnallisuuksia, teknistä tai laadullista velkaa syntyy. Velka muodostuu esimerkiksi arkkitehtonisista kompromisseista, puuttuvista yksikkötesteistä, testaamattomista ohjelmiston osista, korjaamattomista virheistä ja vahvistamattomista virheiden korjauksista. Laatuvelka huomataan usein julkaisuaikajänteellä ta-

[2]Fitnesse on Wiki-integroitu hyväksymistestauskirjasto, josta löytyy lisää tietoja osoitteesta http://fitnesse.org/ [27.3.2008].

pahtuvan laadunvarmistuksen eli talon sisäisenä alfa-testauksen, talon ulkopuolisena beeta-testauksen ja asiakkaan hyväksymistestauksen avulla. Teknisen velan kasvaessa liian suureksi voidaan järjestää erityisiä stabilisaatio- tai testauspyrähdyksiä *(stabilization sprint)* pienentämään teknistä velkaa. Tekninen velka ei ole korotonta, sillä sen suuri määrä hidastaa kehitystyötä. Stabilisaatiopyrähdyksien määrä ja kesto riippuvat kertyneen teknisen velan määrästä, tuotteen kokoamisen ja integraation vaatimasta työstä, virheiden korjausten monimutkaisuudesta ja julkaisun laatukriteereistä [Gal07]. Jokainen näistä voi aiheuttaa tarvetta lisäpyrähdyksille.

3.2.4 Strateginen aikajänne

Aiempien aikajänteiden laadunvarmistus tarvitsee tuekseen tuotteen laadun strategista kehittämistä [IRL05]. Näitä tehtäviä ovat testaajien palkkaaminen, testaajien organisointi ja ohjaus, alemman tason testausmenetelmien onnistumisen arviointi, erilaisten testausympäristöjen tai -järjestelmien rakentaminen sekä laadunvarmistuksen tavoitteiden ja prioriteettien tunnistaminen eli laatusuunnitelman ja testausstrategian laatiminen.

Laatusuunnitelman ja testausstrategian tulee linkittyä Scrumin suunnittelun tasoihin. Tästä ei ole löytynyt kirjallisia suosituksia tai tutkimuksia. Kysely agile-testing@yahoogroups.com -sähköpostikeskusteluryhmään[3] paljasti, että ainakin kaksi yritystä laatii ylemmän tason laatusuunnitelman tai testausstrategian osana tuotteen tiekarttaa tai julkaisusuunnitelmaa, mutta jättää alempien tasojen toteutuksen avoimeksi. Lähestymistapa on ketteryyden mukainen siten, että yksityiskohtia ei suunnitella liian tarkasti, vaan jätetään tilaa reagoida muuttuvaan kehitystilanteeseen. Sähköpostikysely ja vastaukset ovat liitteenä 3.

3.3 Testauksen synkronointi

Testausmenetelmät voidaan sijoittaa aikajänteisiin niiden luonteen perusteella [IRL05]. Sydämenlyönnin aikana tapahtuva testaus on synkronoitua testausta, iteraation aikana tapahtuva testaus osittain synkronoitua. Synkronoimaton testaus ei ole sidottu

[3] Agile-testing@yahoogroups.com -sähköpostiryhmän tarkoitus on keskustella, kuinka testaus tulee järjestää ketterässä ohjelmistoprosessissa. Ryhmän keskusteluun osallistuu säännöllisesti useita ketterän kehityksen ja testauksen tunnettuja asiantuntijoita, kuten Cem Kaner, Brian Marick, Ron Jeffries, Michael Bolton, Matt Heusser, Lisa Crispin sekä lisäksi monia muita alan ammattilaisia ja tutkijoita. Lisää tietoa listasta löytyy osoitteesta: http://www.nabble.com/Agile-Testing-f15311.html [13.3.2008].

mihinkään aikajänteeseen. Testausmenetelmät on sijoitettu aikajänteille taulukossa 3.

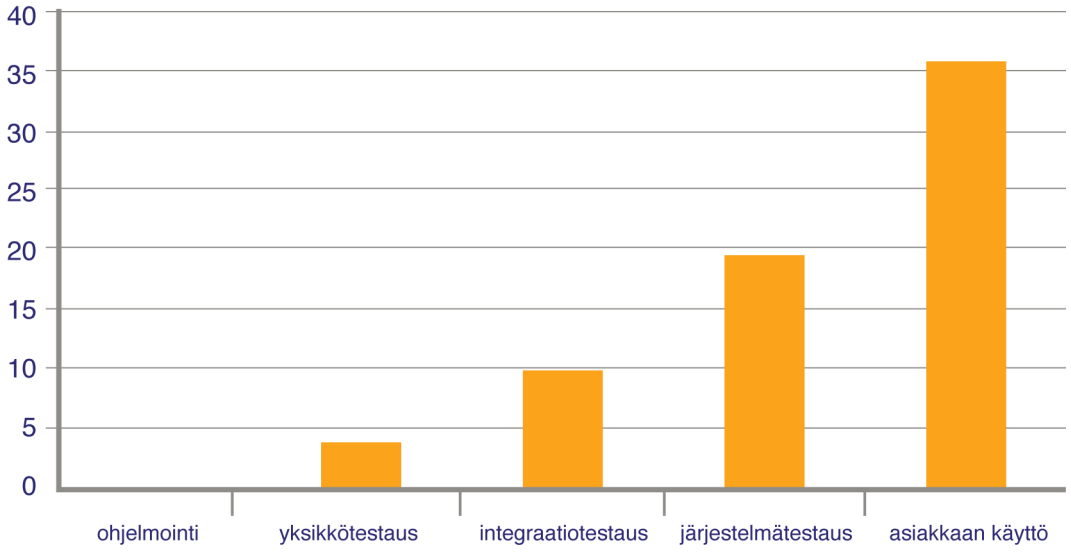
	Sydän	Iteraatio	Julkaisu
Yksikkö	X		
Integraatio	X		
Järjestelmä	X		
Iteraation hyväksymis		X	
Konfiguraatio		X	
Asennus		X	
Suorituskyky		X	
Alfa			X
Beeta			X
Asiakkaan hyväksymis			X

Taulukko 3: Aikajänteet ja testausmenetelmät sijoitettuna taulukkoon, joka kertoo missä aikajänteessä kukin testausmuoto pääsääntöisesti esiintyy.

Ketterät prosessimallit tavoittelevat testauksen synkronointia lyhentääkseen virheen löytymiseen kuluvaa aikaa. Tehokkainta on löytää virhe sydämenlyönnin aikana, sillä kehityksen aikana löydetty virhe on halvin korjata ja aiheuttaa vähiten lisävirheitä [Hie74]. Suurin kustannus aiheutuu virheiden päätymisestä asiakkaalle asti, jolloin niillä saattaa olla suuri merkitys liiketoimintaan. Yleisissä tapauksissa virheiden kustannus jakautuu kuvan 9 osoittamiin vaiheisiin.

Ketterän ohjelmistokehityksen ja testauksen lopullisena tavoitteena on kaiken testauksen synkronointi (in-sync) eli tuominen mahdollisimman lähelle projektin sydämenlyöntiä. Synkronoidussa testauksessa testien tulokset ovat saatavilla muutaman sekunnin kuluessa ohjelmistokoodin kirjoittamisesta [Itk07]. Käytännössä automatisoitukin asennus- tai suorituskykytestaus kestää kymmenistä minuuteista tunteihin ja päiviin. Ajanjaksoa voidaan lyhentää hankkimalla tehokkaampia palvelimia ja suorittamalla muutamia valikoituja testitapauksia useammin.

Manuaalista testausta, kuten alfa- ja beeta-testausta, voidaan nopeuttaa kehittämällä automaatiota. Internetin avulla loppukäyttäjille voidaan lähettää päivityksiä ja testipyyntöjä sekä vastaanottaa virheraportteja. Testitapausten laatiminen helpottuu ja nopeutuu ohjelmointikielien testausominaisuuksien sekä testien laadinta- ja määrittelytyökalujen käytön kautta, jolloin sekä kehittäjät että liiketoimintalogiikkaa tuntevat henkilöt voivat laatia enemmän testitapauksia nopeammin. Samal-

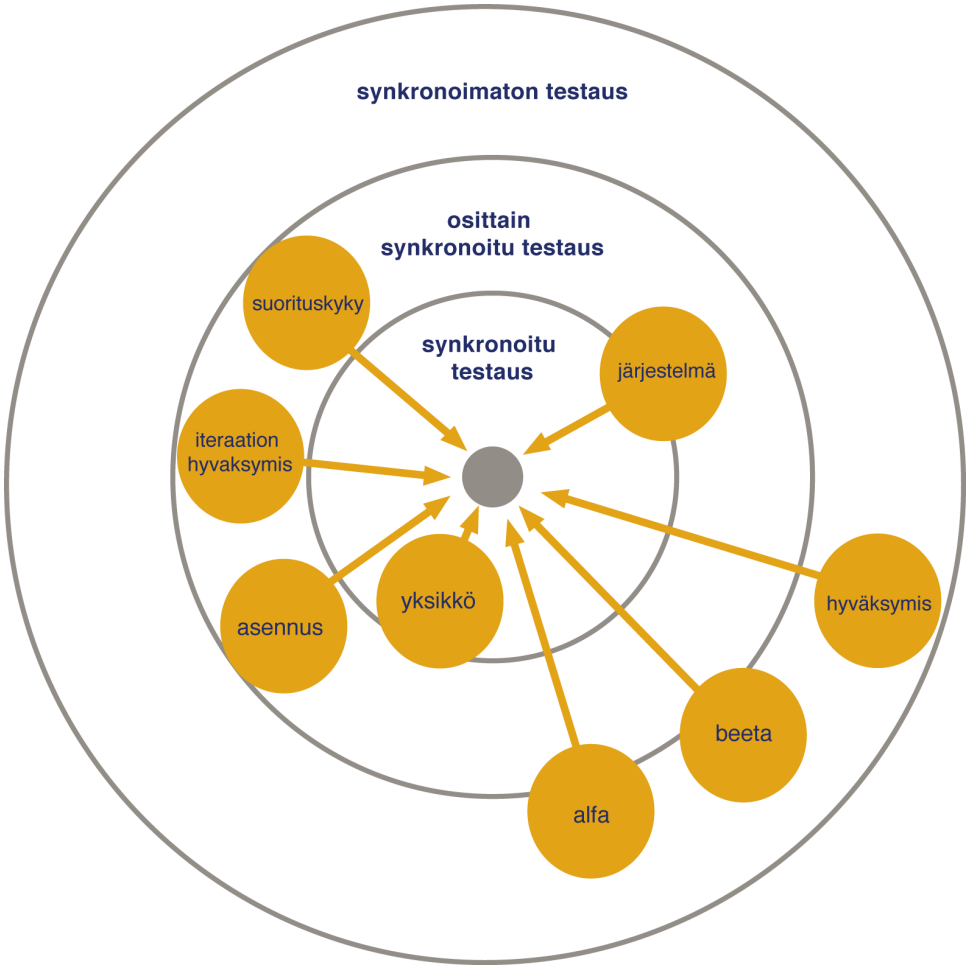


Kuva 9: Virheiden kustannusvaikutus eri vaiheissa [EnR07, s. 18].

la tavoin jokaista testausmenetelmää voidaan nopeuttaa ja siirtää osaksi tai ainakin lähemmäksi synkronoitua testausta. Jokaisen testausmenetelmän täysi synkronointi palvelee parhaiten ketterää tuotekehitystä, mutta se ei ole välttämättä järkevää. Kunkin testausmenetelmän nopeuttamisen hyötyä on arvioitava vasten sen aiheuttamia kustannuksia. Tulevaisuudessa tehokkaammat koneet, helppokäyttöisemmät testausohjelmistot ja ohjelmointikielten kehitys voivat laskea täysin synkronoituun testaukseen siirtymisen kustannuksia ja siten mahdollistaa paremman tuen ketterälle tuotekehitykselle. Kuva 10 havainnollistaa eri testausmuotojen nykyisen sijainnin synkronoinnin tasoilla. Ketterä testaus pyrkii nopeuttamaan jokaista testausmuotoa, minkä vuoksi kuva esittää myös testausmuotojen siirtymisen kohti täysin synkronoitua testausta.

3.4 Testaaja tiimissä

Testaus on ennen kaikkea kommunikaatiota: tietoa virheistä, ohjelmiston ominaisuuksista ja suoritetusta testauksesta. Ketterät menetelmät painottavat kommunikaation tärkeyttä. Testaaja osana ketterää tiimiä on aktiivinen tiedon välittäjä ja vastaanottaja. Käytännössä tämä tarkoittaa aktiivista osallistumista palavereihin ja suunnittelukokouksiin sekä tiedonkulkua edistävien käytäntöjen kehittämistä. Testaajan tulee sekä itsenäisesti että muun tiimin kanssa etsiä tietoa testattavasta ohjelmistosta ja selvittää, mitä kannattaa testata. Ketterällä testaajalla on hyvä olla teknistä osaamista ja kokemusta muun muassa mustalaatikkotestauksesta, testien



Kuva 10: Testausmenetelmien siirtyminen kohti synkroonia.

automatisoinnista, skriptien kirjoittamisesta ja tietokannoista [Cri06a].

Ratkaisevimmiksi ominaisuuksiksi nousevat kommunikaatiotaidot ja joustava asenne. Testaajan tulee olla valmis oppimaan uusia työskentelytapoja ja liittymään tiiviiksi osaksi iteraativista ja itseohjautuvaa tiimiä. Joskus voi olla tarpeellista hämärtää testaajien ja kehittäjien välisiä eroja [Mar04]. Testaajan tulee yhteistyössä kehittäjien kanssa oppia uusia asioita ohjelmistosta ja jopa muokata sitä. Vastaavasti kehittäjän tulee kirjoittaa ja ymmärtää testejä. Ketterissä menetelmissä vastuu laadusta on koko tiimillä. Tiimiin ei välttämättä tarvitse kuulua erikseen nimettyjä testaajia. Riittää, että tiimillä on tarvittavat taidot testauksen suorittamiseksi.

Ketterää testausta voidaan harjoittaa osana perinteisempää prosessimallia. Samat käytännöt kommunikaation ja yhteistyön lisäämisestä ovat hyödyksi myös siellä [Mar01].

4 Tutkimus

4.1 Tutkimuskysymys

Nykyisessä ohjelmistokehityksessä ovat ketterät menetelmät suosittuja. Ketterät menetelmät suosivat XP-lähtöisiä laatukäytäntöjä, yleisimmin tiimit käyttävät ainakin TDD -menetelmää. Prosessimallit eivät kuitenkaan anna selkeää tai yksityiskohtaista vastausta siihen, kuinka paljon muuta testausta kuin yksikkötestausta tulee tehdä, kuinka testausta tulee suunnitella osana tuotteen tulevaisuuden suunnittelua ja kuinka testaus tulee organisoida. Useilla ohjelmistoyrityksillä on jo vuosien positiivinen kokemus ketteristä menetelmistä suosituimman eli Scrumin käytöstä. Näiden yritysten on täytynyt käytännössä ratkaista testauskysymys jollain tavalla. Tutkimus pyrki löytämään käytännön näkemyksiä ja kokemuksia laadun, testauksen ja Scrum-prosessimallin yhteensovittamisesta.

Tutkimus keskittyi Scrum-prosessimalliin, jotta voitiin selvittää asioita yksityiskohtaisesti ja tunnistaa laatutoimenpiteiden suhde Scrumin käytäntöihin. Tutkimuksen tekijä työskenteli testauksen kehittämisen parissa Scrum-prosessimallia käyttävässä yrityksessä. Tästä lähtökohdasta oli helpompi miettiä tutkimusasetelmaa ja -järjestelyjä etukäteen, suunnitella kysymysten aihepiirejä ja lähestyä yrityksiä ammattimaisesti.

4.2 Yritysten valinta

Tutkimukseen haettiin yrityksiä, jotka sijaitsevat pääkaupunkiseudulla ja joilla on omaa tuotekehitystä. Lisäksi yrityksien edellytettiin työllistävän 40-100 henkeä ja saaneen vähintään muutaman vuoden kokemuksen Scrumin käytöstä.

Potentiaalisiin yrityksiin lähetettiin sähköpostikysely halukkuudesta osallistua tutkimukseen (liite 1). Kyselyjä lähetettiin seitsemään yritykseen. Yksi yritys (yritys A) vastasi ensimmäiseen kyselyyn, toinen yritys (yritys B) vastasi myöntyvästi toiseen tiedusteluun ja kolmas suostui mukaan vasta puhelinsoiton jälkeen. Kolmannella yrityksellä oli kuitenkin haasteita aikataulujensa kanssa. Alkuhaastattelun jälkeen tarvittavien lisähaastattelujen sopiminen muodostui vaikeaksi ja lopulta haastateltavissa olisi ollut vain yksi henkilö tarvittavan kolmen sijaan. Tämän vuoksi yritys päätettiin jättää tutkimuksen ulkopuolelle. Neljättä yritystä lähestyttiin työntekijäkontaktin avulla, jolloin saatiin tietää teknistä kehitystä johtavan henkilön nimi. Hän ei kuitenkaan vastannut sähköpostitiedusteluihin tai toistuviin puhelinsoittoi-

hin. Viides, kuudes ja seitsemäs yritys eivät vastanneet sähköpostikyselyyn lainkaan. Tutkimukseen osallistui lopulta kaksi yritystä. Tutkimuksen kannalta olisi ollut parempi saada mukaan useampia yrityksiä laajemman vertailupohjan saavuttamiseksi.

Syyt yritysten kiireisiin tai vastaamatta jättämisiin voivat olla monet. Paras vinkki tulee kolmannen yrityksen toiminnasta. Testausta johtava ja kehittävä henkilö suhtautui positiivisesti tutkimuksen tekoon, mutta tutkimuksen teko vaikeutui hänestä riippumattomien sisäisten haasteiden takia. On todennäköistä, ettei yrityksessä uskallettu osallistua tutkimukseen, koska omat laatukäytännöt olivat heikot eikä niitä haluttu kehittää eikä ikäviä totuuksia kohdata. Myös kova kiire saattoi estää halun kehittää ja parantaa toimintaa. Ehkä näistä syistä suurin osa yrityksistä jätti vastaamatta tutkimuskyselyyn. Tätä olettamusta tukee ensimmäisen ja toisen yrityksen positiivinen asenne tutkimukseen, halu oman toiminnan kehittämiseen sekä yritysten prosessien ja laadun kohtalaisen hyvä tilanne verrattuna vaikutelmaan kolmannen yrityksen tilanteesta. Ensimmäisessä ja toisessa yrityksessä työskentelevät laadusta vastuulliset henkilöt uskoivat tilanteensa olevan hyvä ja uskaltautuivat siksi laatutoimiensa analyysiin ja mahdolliseen arvosteluun. Muut syyt vastaamattomuuteen ja tutkimuksesta kieltäytymiseen voivat olla inhimillinen erehdys, usko omaan laatuun ja epäily tutkimuksen tekijän pätevyydestä sekä tutkimuksen hyödyllisyydestä.

4.3 Aloituspalaveri yrityksissä

Suostumuksen jälkeen yritysten kanssa sovittiin sähköpostilla ja puhelimella aloituspalaveri, jossa tutkimuksen tekijä esitteli hankkeensa tuotekehityksestä tai testauksesta vastaavalle johtajalle. Palaverissa sovittiin lopullisesti tutkimuksen suorittamisesta yritykseen, valittiin sopiva tiimi ja tuote tutkimusta varten ja sovittiin osittain tutkimuksen aikatauluista. Samalla käytiin lyhyesti läpi yrityksen toimiala, tuotteet, asiakkaat, käytetty teknologia, organisaatio, käytössä olevat kehitystyökalut, johtajan näkemys tilanteesta ja kehityksen tarpeesta sekä tutkimuksen käytännön järjestelyihin liittyviä asioita. Lisäksi allekirjoitettiin salassapitosopimukset tutkimuksen tekijän ja yrityksen välillä. Palaverin yhteydessä haastattelija esiteltiin osalle haastateltavistaan ja sovittiin haastattelujen aikatauluista. Osittain haastattelujen aikatauluista sovittiin sähköpostitse. Haastattelija teki muistiinpanoja aloituspalaverin aikana, otti kuvia tauluille piirretyistä kaavioista ja sai mukaansa yritystä ja sen tuotekehitystä yleisesti esittelevää materiaalia. Aloituspalaverien ilmapiiri oli positiivinen - tekniset johtajat olivat motivoituneita ja kiinnostuneita organisaationsa

kehittämisestä ja uusista ideoista. Syynä tähän lienee mahdollisuus saada ilmaiseksi vertailutietoa muista alan yrityksistä ja analyysi oman toiminnan tilasta ja kehittämisestä ulkopuolisen silmin. Aloituspalaverit olivat yritysten neuvotteluhuoneissa 19.10.2007 ja 29.10.2007.

4.4 Haastattelujen järjestelyt

Haastatteluihin tavoiteltiin neljän roolin edustajia; tuotteen omistajia, Scrum-mestareita, kokeneita kehittäjiä ja testauksesta vastaavia. Näiden roolien haltijoille kertyy kokemusta kehitystyöstä ja roolien kautta käytetään paljon valtaa. Tuotteen omistajan valta tulee Scrum-prosessimallin mukaisesti siitä, että hän päättää iteraation toimenpiteiden tärkeysjärjestyksen ja vastaa tuotteen pidemmän aikavälin suunnittelusta. Laadun tulee olla hänelle tärkeä ja tuttu työväline. Scrum-mestari ei ole johtaja, vaan valmentaja. Hänellä on näkemystä prosessimallin oikeasta ja väärästä toteuttamisesta. Hän kykenee havaitsemaan, mikä prosessin toteuttamisessa on mahdollisesti vialla ja miten laatu ja prosessimalli voidaan sovittaa yhteen. Kokeneilla kehittäjillä on paljon valtaa teknisen kyvykkyyden ja tuotteen syvällisen tuntemuksen vuoksi tiimin sisällä ja ulkopuolella. Heillä on pitkä kokemus tuotteen laadun kehittymisestä. Testauksesta vastaava joutuu päivittäisessä työssään sovittamaan Scrumin ja laadunvarmistuksen yhteen. Hän tietää, mitä testataan, mitä pitäisi testata ja kuinka prosessia tulee kehittää huomioimaan laatu ja testaus paremmin. Usein kuitenkin yrityksen sisäpolitiikka estää parannukset prosessiin; testauksen ja laadun oikeita painotuksia ja tarvetta on vaikeampi määrittää ja perustella liiketoiminnan näkökulmasta kuin uusien ohjelmapiirteiden.

Haastattelujen rakenne oli kaikille rooleille samanlainen ja jokaiselle esitettiin samat kysymykset testauksesta ja tuotantoprosessista. Kunkin roolin kohdalla painotettiin eri kysymyksiä. Scrum-mestarilta kysyttiin erityisesti Scrum-prosessin laadusta ja erikoisuuksista sekä laadunvarmistuksesta prosessin kannalta. Tuotteen omistajan kohdalla painotettiin näkemyksiä ja kokemuksia tuotteen laadunvarmistuksesta pidemmällä tähtäimellä sekä omasta roolista ja vastuusta laadun suhteen. Testaajaa haastateltiin erityisesti omasta roolista, työkuvasta ja työtehtävistä. Kokeneelta kehittäjältä kysyttiin omasta roolista laadunvarmistuksessa. Roolit tiimeissä lomittuivat, ja kummastakin haastateltiin kolmea henkilöä. Tiimi A:ssa toisen osatiimin testaaja toimi Scrum-mestarina ja tiimi B:ssa ei ollut nimettyä Scrum-mestaria. Kysymykset ja painotukset ovat liitteessä 2.

Haastattelut etenivät varsin vapaasti ja syventyivät esille nousseisiin mielenkiintoi-

siin aiheisiin. Haastattelun lopuksi tarkastettiin, oliko kaikki aiheet käyty läpi ja tarvitsiko jokin vielä lisävalaistusta. Aiemmat haastattelut auttoivat syventämään keskustelua ja löytämään uusia aiheita. Haastattelut nauhoitettiin haastattelutilanteen sujuvuuden varmistamiseksi. Haastattelut olivat muutoin henkilökohtaisia, mutta tiimin B osalta haastateltiin aikataulujen vuoksi tuotteen omistajaa ja kokenutta kehittäjää samaan aikaan. Henkilökohtaiset haastattelut kestivät 43-54 minuuttia. Parihaastattelu kesti yhden tunnin ja yhdeksän minuuttia. Haastattelut suoritettiin yritysten neuvotteluhuoneissa 30.10.-13.11.2007. Haastattelujen jälkeen auki jääneitä asioita tarkennettiin puhelimen ja sähköpostin avulla.

4.5 Haastattelujen onnistuminen

Käytännön järjestelyt sujuivat juohevasti, ja haastattelut olivat rentoja ja luovia tilanteita. Luontevimmin omasta työstään puhuivat kokeneemmat työntekijät. Uudemmilla tulijoilla haastattelutilanne aiheutti ainakin aluksi hieman hapuilua, epävarmuutta ja työväsymyksen tuottamaa ärtyneisyyttä. Jotkut saattoivat kokea haastattelutilanteen kyseenalaistavan heidän ammattitaitoaan ja tekemäänsä työtä. Tämä korostui erityisesti haastateltaessa testauksesta vastaavia henkilöitä. Nämäkin haastateltavat kuitenkin rentoutuivat keskustelun edistyessä. Haastateltavat ja yritysten edustajat olivat lähes poikkeuksetta hyvin motivoituneita tutkimukseen. Useimmiten päästiin keskustelemaan suoraan kehityksen hyvistä ja huonoista puolista.

Haastattelujen suurimpana heikkoutena on niiden subjektiivisuus. Haastateltavia johdateltiin aihepiiriä kuvaavilla yleisillä kysymyksillä ja sallittiin siirtyminen eri aihepiiristä toiseen vapaasti. Vasta lopussa tarkistaen että kaikki osa-alueet on käyty läpi. Tarkkoja kysymyksiä ei pystytty laatimaan, koska se olisi vaatinut liian mittavaa tutustumista yrityksiin. Ilman tarkkoja kysymyksiä haastateltavat pystyivät väistämään hankalia aihealueita, jollei haastattelija huomannut painottaa asiaa.

4.6 Toimitettu materiaali

Haastattelujen lisäksi yritykset toimittivat teknistä dokumentaatiota, kuten laatusuunnitelmiaan, testaussuunnitelmia, kattavuusanalyysejä, kehittämissuunnitelmia ja prosessin tilaa kuvaavia tunnuslukuja. Materiaali oli kuitenkin hyvin sisältöriippuvaista, joten pitkälle vieviä johtopäätöksiä ei siitä kyetty tekemään. Materiaalista hyödyllisimpiä olivat yksikkötestien kattavuusanalyysit, joiden avulla oli helppo

arvioida yksikkötestien laajuutta ja määrää. Muu dokumentaatio antoi hyvän kuvan toiminnan kehittämisestä yrityksessä. Pitkälle vietyjen laatusuunnitelmien tai testausstrategioiden puuttuminen oli selkeä merkki puutteista laadun suunnittelussa ja toteutuksessa. Materiaalin analyysi oli tarkoitettu pääsoin haastattelujen teon ja analysoinnin tueksi. Materiaalin tarkempi analysointi ja johtopäätösten tukeminen siihen ei näin sisältöriippuvaisessa tutkimuksessa olisi ollut järkevää.

5 Tutkimuksen tulokset

Yritys A työllistää 130 henkeä, joista 70 henkeä työskentelee tuotekehityksessä. Yritys tuottaa internetin kautta käytettäviä tuotteita ja niiden tukituotteita. Tuotteet käsittelevät rahaa, mikä lisää niiden korkean laadun tarvetta. Yrityksellä on yksi pääasiallinen asiakas, joka on samalla yrityksen pääomistaja. Pääasiakas sijaitsee Suomessa. Muut asiakkaat ovat Suomessa ja ulkomailla. Jokaisella osatuotteella on oma tuotantotiimi, joista osa käyttää Scrumia ja osa perinteisiä kehitysmenetelmiä. Tuotantotiimien tuotokset yhdistetään yhtenäiseksi tuotteeksi integraatiotiimin toimesta kuusi kertaa vuodessa. Eri tuotantotiimien iteraatiot on synkronoitu samaan tahtiin, mikä estää yksittäistä tiimiä siirtymästä sille mahdollisesti paremmin sopivaan iteraation pituuteen. Yrityksen testausta johdetaan keskitetysti, mutta testaajat on hajautettu tiimeihin. Tutkimukseen valittiin Scrumia käyttävä tiimi (tiimi A).

Yritys B työllistää 65 henkeä, joista 20 henkeä on tuotekehityksessä. Yrityksellä on kaksi päätuotetta, jotka perustuvat saman teknologian käyttöön, mutta joiden käyttötarkoitus on eri. Yrityksellä on useita asiakkaita sekä Suomessa että ulkomailla. Tuotekehitys koostuu kahdesta eri tiimistä, joissa kummassakin on tiimin johtaja, viisi kehittäjää ja kaksi testaajaa. Tutkimukseen valittiin mukaan toinen tiimeistä (tiimi B).

Yritys A oli kooltaan suurempi kuin tavoiteltu kohderyhmä, mutta muuten yritykset sopivat hyvin tutkimuksen tavoitteisiin.

	yritys A	**yritys B**
työntekijöiden lukumäärä	130	65
tuotekehityksen lukumäärä	70	20
tuotantotiimien lukumäärä	7	2

Taulukko 4: Yritysten ominaisuudet.

Tiimit A ja B olivat rakenteeltaan hyvin samanlaisia, ja erityisesti kehittäjien ja testaajien suhde ja roolijako oli samanlainen. Molempien tiimien testaajat selvisivät välttävällä tavalla pyrähdyksen aikana lisättävien uusien ohjelmistopiirteiden manuaalisesta testauksesta. Aikaa pidemmän tähtäimen suunnitteluun, testauksen kehitykseen tai esimerkiksi automaattisten järjestelmätestien kirjoittamiseen ei ollut.

Tiimi A erosi tiimistä B sijaitsemalla maantieteellisesti kahdessa eri paikassa. Koh-

teessa A1 oli neljä kehittäjää ja yksi testaaja ja kohteessa A2 oli kolme kehittäjää ja yksi testaaja. Tuotteen omistaja sijaitsi kohteessa A1. Iteraation lopussa toinen osatiimi matkusti toisen luokse. Tiimi A oli käyttänyt Scrumia noin kaksi vuotta ja käytti neljän viikon pyrähdyksiä. Tiimi A:n kohdalla lyhyempään pyrähdykseen siirtyminen olisi aiheuttanut jaon takia kohtuuttoman suuria ajallisia kustannuksia, koska yhteiset palaverit matkustusaikoineen olisivat vieneet huomattavan osan pyrähdykseen käytössä olevasta ajasta. Tiimillä A oli käytössään Scrum-mestari. Haastateltavien vastausten mukaan uusien ohjelmistopiirteiden paine tuntui olevan liian kova.

Tiimi B koostui tiimin johtajasta, viidestä kehittäjästä ja kahdesta testaajasta. Tiimi B oli käyttänyt Scrumia kolme vuotta ja käytti yhden viikon pyrähdyksiä. Haastateltavien vastausten perusteella yhden viikon pyrähdykset tuntuivat toimivan paremmin kuin tiimi A:n pidemmät pyrähdykset, mutta varmuutta toimivuudesta ei haastattelujen perusteella ole. Lisäksi tiimi B:n jäsenet olivat motivoituneempia lyhyiden pyrähdyksien käyttöön. Tiimi B:n pidempi kokemus Scrumista viittaisi siihen, että pyrähdyksien pituutta lyhentämällä saavutetaan tehokkaampi prosessi, mutta tätä ei voida vahvistaa pelkkien haastatteluvastausten perusteella. Tiimi B:n kohdalla Scrum-mestarin tehtävät oli jaettu yhteisvastuullisesti tiimin jäsenten kesken. Uusien ohjelmistopiirteiden paine oli jonkinasteinen ongelma myös tiimi B:n kohdalla.

	tiimi A	**tiimi B**
pyrähdyksen pituus	4 viikkoa	1 viikko
Scrum-mestari	on	ei ole
tiimin jäseniä	9	8
testaajia tiimissä	2 (22 %)	2 (25 %)
Scrumia käytetty tiimissä	2 vuotta	3 vuotta
muuta	jaettu kahteen paikkaan	

Taulukko 5: Tiimien ominaisuudet.

5.1 Testauksen strateginen suunnittelu

Yritys A oli laatinut testauksen kehittämisprosessin, mutta yritykseltä puuttui laatusuunnitelma ja tuotekohtainen testausstrategia. Lisäksi tuotteen A testausstrategia oli työn alla. Käynnistetty kehittämisprosessi antoi strategiatason näkemystä ja ohjausta prosessin kehittämiseksi. Kehittämisprosessi oli käynnistetty kuudesta

syystä (taulukko 6). Kehittämisprosessi määrittelee testauksen tavoitteiksi tuottaa laatutietoa julkaisupäätöksen teon tueksi, löytää tärkeät virheet mahdollisimman aikaisin testausprosessissa ja auttaa kehittäjiä löytämään ongelmakohtia tuotteista. Ongelmien ratkaisemiseksi ja tavoitteiden saavuttamiseksi laatu tulee rakentaa sisään tuotteeseen alusta alkaen. Käytännön toimet tavoitteiden saavuttamiseksi on esitetty taulukossa 7. Toteutuessaan kehittämisprosessi tuottaisi tarvittuja parannuksia yrityksen prosesseihin.

ongelmat
Tuotteiden huono laatu, joka aiheutti paljon työtä ongelmien ilmettyä vasta tuotannossa.
Laatutiedon puute, joka johtui vähäisestä alhaisen tason funktionaalisesta mustalaatikkotestauksesta, kunnollisen integraatiotestauksen puuttumisesta ja epäselvistä testauksen raportointikäytännöistä.
Virheiden löytyminen testauksen myöhäisessä vaiheessa eli järjestelmä- tai hyväksymistestauksen aikana. Tämän johdosta suuri osa testauksen työmäärästä kului järjestelmätestauksen tasolla.
Tuotanto-organisaation asenne, jonka mukaan integraatiotiimi ja sen tekemä testaus toimi hyvänä turvaverkkona ja ongelmat voidaan surutta jättää sen hoidettaviksi.
Huonosti ja epäyhtenäisesti suunniteltu testaus. Puutteita löytyi muiden muassa suunnittelusta ja tulosten sekä virheiden raportoinnista.
Testausprosessin huono yhteensopivuus Scrumin kanssa eli testaus ei ollut läpinäkyvää eikä hyväksymiskriteereistä pidetty kiinni.

Taulukko 6: Yrityksen A kehittämisprosessin havaitsemat ongelmat.

toimenpiteet
Testaajia ei tule käyttää varmuusverkkona.
Eri tiimien käyttämät tuotantojärjestelmät tulee yhtenäistää.
Tulee rakentaa yhteinen integraatioympäristö.
Testaus tulee saada nykyistä synkronoidummaksi osaksi Scrum-prosessia.
Osa manuaalisesta testauksesta tulee ulkoistaa.
Valmiin ohjelmistopiirteen määrittelyä tulee tarkentaa.

Taulukko 7: Yrityksen A kehittämisprosessin ratkaisut ongelmiin.

Yrityksestä B ei löytynyt laatusuunnitelmaa eikä testausstrategiaa. Halua toiminnan

kehittämiseen löytyi tekniseltä johtajalta, kehittäjiltä, testaajilta ja tuotteen omistajalta. Järjestelmällistä kehittämistoimintaa ei ollut. Syynä tähän voi olla kehittämisen eteneminen osana muuta prosessia ja yrityksen tuotekehityksen kohtuullisen hyvä tila. Pieni koko voi myös helpottaa asioiden edistämistä ilman laajoja toimenpideohjelmia. Kuitenkin jatkuva ja tavoitteellinen kehitys vaatii tuekseen suunnitelmia ja niiden noudattamista.

5.2 Automatisoitu testaus

Yksikkötestien lausekattavuus oli tuotteessa A 20% ja tuotteessa B 46%. Tiimi A pitikin suurimpana ongelmanaan *perinnekoodia (legacy-code)*, jonka laatu oli heikkoa ja jolle ei ollut yksikkötestejä. Kumpikin tiimi kirjoitti yksikkötestejä osana ohjelmistokehitystä. Testien kirjoittamista ei seurattu keskitetysti. Yksikkötestien määrälle ei ollut asetettu tavoitearvoja esimerkiksi kattavuuden avulla. Erityinen huoli tiimi A:n kohdalla oli, että huonon laadun riski kasvaa yksikkötestien alhaisen kattavuuden seurauksena.

Muiden automatisoitujen testien määrä oli kummallakin tiimillä pieni. Käytössä oli muutamia järjestelmätason funktionaalisia testejä ja suorituskykytestejä. Pienen määrän johdosta testit eivät hyödyttäneet kehitysprosessia juuri lainkaan. Automatisoidun testauksen laaja kirjo jäi puuttumaan.

Testauksen pieneen määrään on kaksi syytä: käytössä olevien resurssien vähyys ja epäselvyys siitä, mihin testauksen lisäresurssit kannattaa kohdentaa. Resursseja on vähän, koska yrityksen johto, tuotteen omistaja tai asiakas ei ymmärrä testauksen merkitystä laadun selvittämiseksi ja parantamiseksi. Tämä selvisi kummankin yrityksen tuotteen omistajien ja laatuvastaavien haastatteluista; heidän itsensä vaikea mitata laadun vaatimien toimenpiteiden hintaa tai perustella sitä ylemmille tsaoille. Resurssitarpeen esille tuominen on tiimin ja laatuvastaavan vastuu. Laadun ja testauksen suunnitelmat tulee johtaa yrityksen liiketoiminnan strategisista tavoitteista, jolloin testauksen resurssien lisäämiselle voidaan löytää järkeviä syitä ja lisäresurssit voidaan perustella liikkeenjohdolle.

5.3 Tiimin toiminta ja Scrum

Kummallakin tiimillä tuntui olevan vähän mahdollisuuksia laadun vaatimien toimintojen saamiksesi osaksi pyrähdyksiä. Taitoa ja kokemusta laadusta huolehtimiseen

tiimien jäsenillä on, mutta ei aikaa. Periaatteessa tiimi on vastuussa laadusta ja voisi keskittyä tekemään laatua edistäviä toimenpiteitä pyrähdyksessään, mutta käytännössä liiketoiminnan näkökulma uusien piirteiden lisäämiseksi ajaa tämän ylitse. Tiimi B oli yrityksen hieman pienemmän koon johdosta paremmassa asemassa tekemään merkittävämpiä laadunparannustoimenpiteitä kuin tiimi A. Lisäksi tiimi B oli Scrum-prosessimalliin siirtymisensä yhteydessä saanut virheiden määrän vähentymään ja uusien piirteiden lisääminen oli nopeutunut. Siltikin uuden ohjelmistopiirteen valmiiksi merkitsemisen kriteerit olivat molemmilla tiimeillä epämääräiset ja löysät.

Molemmissa tiimeissä roolijaot kehittäjien ja testaajien välillä olivat tiukat. Testaaja on usein tiimissä aliarvostettu eikä hänellä ole vaikutusvaltaa tiiminsä jäseniin. Monet hyvät kehittämisajatukset voivat siten jäädä huomiotta.

Kumpikin yritys käytti tuotteen omistajana oikean asiakkaan sijasta yrityksen organisaatiosta valittua henkilöä. Tämä etäännytti ketteryyden periaatteiden vastaisesti tiimit asiakastodellisuudesta. Oikeiden asiakkaiden käyttö on usein vaikeaa esimerkiksi asiakkaiden suuren määrän tai etäisyyksien johdosta. Tällaisessa tapauksessa tiimin tulee huolehtia asiakkaan tarpeiden ja näkemysten kohtaamisesta muiden keinojen avulla, kuten asiakasvierailuilla, hyvillä asiakkaiden virheraportointi- ja tukipalveluilla sekä asiakaskyselyillä.

5.4 Suositukset yrityksille

Haastattelujen jälkeen kummallekin yritykselle annettiin yrityskohtaiset suositukset toiminnan parantamiseksi. Yritys A pyysi ja sai suosituksensa taulukkomuodossa työn ja hyödyn mukaan lajiteltuina. Samanlainen taulukko tehtiin myös yrityksen B suosituksista. Tutkielman havainnot ja suositukset esiteltiin yrityksille palavereissa, joissa keskusteltiin jatkotoimista. Suosituksista esitellään yleiset huomiot, taulukkomuotoiset yhteenvedot ja niiden perusteella yrityksissä päätetyt toimenpiteet. Kehittämisehdotusten puuttuvat osat ovat liitteinä 4 ja 5.

5.4.1 Yritys A

Tiimi A oli hieman ylityöllistetty, ja sen pahin ongelma oli jakautuminen kahteen toimipisteeseen. Etäisyyttä lisäsi osatiimien jäsenien eri äidinkielet, vaikka yhteisenä kehityskielenä käytettiin englantia. Maantieteellisen ja kulttuurillisen etäisyyden voittamiseen kuluva ylimääräinen aika heikensi tiimin jäsenten motivaatiota. Scru-

min painottama välitön tiedonkulku kärsi, vaikka käytössä oli teknisiä apuvälineitä, kuten Skype, videokokoukset ja chat. Toisen osatiimin sijainti osana muun yrityksen pääkonttoria saattoi aiheuttaa eräänlaisen isäntä-renki-asetelman, jota kieli- ja kulttuurimuuri vahvisti.

Huonon laadun riskitekijöitä olivat yksikkötestauksen varmennuksen ulkopuolella olevan perinne- ja muun koodin suuri määrä ja uusien ohjelmistopiirteiden valmistumisen kova kiire. Riskit rahaa käsittelevän ohjelmiston tuotannossa tuntuivat suurilta ja siksi laadunvarmistustoimintoihin tuleekin panostaa enemmän. Tällaisessa tilanteessa on järkevimpiä selviytymistaktiikoita käyttää Scrumin kaltaista joustavaa prosessimallia.

Taulukossa 8 on arvioitu kehittämiskohteita hyödyn ja työn määrien suhteen. Työmäärien ja hyötyjen tarkkojen määrien arvioiminen on vaikeaa, jonka vuoksi ne on arvioitu suhteessa toisiinsa. Tarkka arviointi edellyttäisi tarkempaa tuntemusta yrityksistä, yksityiskohtaisempien suunnitelmien laatimista ja hyödyn määrää selvittäviä tarkkoja analyyseja. Tarkka arviointi on rajattu tutkielman ulkopuolelle tutkimuskysymyksen kannalta epäolennaisena ja suuren työmääränsä takia. Taulukossa 9 on esitelty kehittämisehdotukset yksityiskohtaisesti.

	suuri hyöty	**normaali hyöty**	**pieni hyöty**
suuri työ	kokonaistuote		
normaali työ		yksikkötestaus, järjestelmätestaus	
pieni työ		testausstrategia	hyväksymiskriteerit, lyhyt pyrähdys

Taulukko 8: Yrityksen A kehittämiskohteet suhteutettuna työn ja hyödyn määrän suhteen.

nimi	**ongelmakohta**	**parannusehdotus**	**selitys/positiiviset seuraukset**
kokonaistuote	Liian pitkä julkaisusykli, jolloin paljon virheitä siirtyy julkaisuprosessin loppupuolelle. Virheet ovat kalliita korjata, ja ne saattavat päätyä asiakkaalle asti.	Osatuotteista koostuvan kokonaistuotteen nopeampi automaattinen rakentaminen, asennus ja automatisoitu testaus.	Nopeampi palautesykli virheille, integraatiovirheiden määrä vähenee ja parempi tiedonkulku eri tiimien välillä.
yksikkötestaus	Yksikkötestien kattavuus on huono, jolloin tuotekehitys on virhealtista ja tehotonta.	Yksikkötestien kirjoittaminen ja koodin refaktorointi osana jokaista pyrähdystä.	Virheiden määrä vähenee ja uusien piirteiden lisääminen nopeutuu, eli tuotekehitys tehostuu.
järjestelmätestaus	Järjestelmätason testejä ei juuri ole. Tuotekehitys on virhealtista ja tehotonta.	Automaattisten järjestelmätestien kirjoittaminen jokaisessa sprintissä.	Virheiden määrä vähenee ja uusien piirteiden lisääminen nopeutuu, eli tuotekehitys tehostuu.
testausstrategia	Tuotteen laatu ei parane, työskentely ei ole tavoitteellista.	Testausstrategian ja laatusuunnitelman laatiminen tuotteelle.	Laatutoimenpiteet toteutuvat suunnitellusti ja resurssien sisällä.
hyväksymiskriteerit	Laadunvarmistus jää usein tekemättä uusien piirteiden lisäämisessä.	Hyväksymiskriteerien tiukentaminen.	Testit ja dokumentaatio tulevat kirjoitetuksi. Varmistutaan siitä, että uusi piirre on asiakkaan haluama ja oikea.
lyhyt pyrähdys	Liian pitkä pyrähdys ei fokusoidu tarpeeksi, uusien piirteiden lisäämisessä jää usein laadunvarmistus tekemättä.	Sprinttien lyhentäminen 1–2 viikkoon.	Tarkempi fokus uusille piirteille, jolloin laatu ja oikeellisuus tulee huomioitua tarkemmin.

Taulukko 9: Yrityksen A kehittämiskohteet.

5.4.2 Yritys A:n johtopäätökset suosituksista

Yrityksen A kanssa pidettiin palautepalaveri tutkimuksen tuloksista 26.3.2008. Palaveriin osallistui 12 henkeä yrityksen tuotekehityksestä, osa videoneuvotteluyhteyden takaa. Palaverissa esiteltiin tutkimuksen yleiset tulokset ja käytiin yrityksen kehittämisehdotukset läpi. Keskustelu palaverissa oli vähäistä, mutta pääosin tuloksia ja havaintoja myötäilevää. Kriittistä keskustelua herätti erityisesti ehdotus lyhyemmän pyrähdyksen käyttöönotosta. Yrityksen seitsemän tuotantotiimiä on synkronoitu toimimaan samassa neljän viikon iteraatiotahdissa, jolloin yhden tiimin siirtäminen aiheuttaisi muutoksia muihin.

Yrityksen ja tiimin edustajia pyydettiin kertomaan tutkimuksen tulosten ja suositusten aiheuttamista toimenpiteistä. Kysymys ja yrityksen vastaukset ovat kokonaisuudessaan liitteessä 6. Yrityksen edustajien mukaan kaikkiin esille nostettuihin ongelmiin oltiin puuttumassa. Erityisesti taulukossa 10 esitetyt kohdat oli päätetty toteuttaa.

nro.	**toimenpide**
1.	Laaditaan yritykselle testausstrategia.
2.	Asiakas on jo poistanut kehityspaineen tiimin yltä.
3.	Järjestelmätestauksen automatisointi on hyvässä vauhdissa.
4.	ATTD tulee käyttöön.
5.	Liian vahvaa roolitusta testaajien ja kehittäjien välillä ollaan purkamassa.
6.	Tuotteen työlistan käyttöä tarkennetaan.
7.	Tuotteen julkaisu tehdään kolme viikkoa kestävien kehitys- ja stabilisaatiopyrähdyksien jälkeen.

Taulukko 10: Yrityksen A toimenpiteet kehittämisehdotusten johdosta.

5.4.3 Yritys B

Yhden viikon pyrähdykset olivat toimivia, mutta samalla yritys sokeutui pidemmän tähtäimen laadunvarmistukselle. Nopeasyklisyys vaikutti toimivalta ja hyvältä. Testaajat olivat liian kiireisiä ja selvisivät jotenkuten päivittäisistä, lähinnä manuaalisista, testausrutiineista. Testaus tarvitsi kehittyäkseen lisää resursseja. Laadun ja testauksen suunnitelmat tulee johtaa yrityksen liiketoiminnan strategisista tavoitteista, jolloin testauksen resurssien lisäämiselle voidaan löytää järkeviä perusteita.

Taulukossa 11 on arvioitu kehittämiskohteita hyödyn ja työn määrien suhteen. Työmäärien ja hyötyjen tarkkojen määrien arvioiminen on vaikeaa, jonka vuoksi ne on arvioitu suhteessa toisiinsa. Tarkka arviointi edellyttäisi tarkempaa tuntemusta yrityksistä, yksityiskohtaisempien suunnitelmien laatimista ja hyödyn määrää selvittäviä tarkkoja analyyseja. Tarkka arviointi on rajattu tutkielman ulkopuolelle tutkimuskysymyksen kannalta epäolennaisena ja suuren työmääränsä takia. Taulukossa 12 on esitelty kehittämisehdotukset yksityiskohtaisesti.

	suuri hyöty	**normaali hyöty**	**pieni hyöty**
suuri työ			
normaali työ	manuaalinen testaus, automatisoitu testaus	Scrumin kehittäminen	
pieni työ	testausstrategia		

Taulukko 11: Yrityksen B kehittämiskohteet suhteutettuna työn ja hyödyn määrän suhteen.

5.4.4 Yritys B:n johtopäätökset suosituksista

Yrityksen B kanssa pidettiin palaveri tutkimuksen tuloksista 29.2.2008. Palaveri oli alun perin suunniteltu syksylle, mutta ajankohta siirtyi tuotteen julkaisukiireiden takia. Palaveriin osallistui 9 henkeä yrityksen tuotekehityksestä. Palaverissa esiteltiin tutkimuksen yleiset tulokset ja käytiin kehittämisehdotukset läpi. Palaverissa vallitsi hyvä tunnelma ja aiheista keskusteltiin puolentoista tunnin aikana avoimesti. Palaverin pohjalta yritys listasi taulukossa 13 nähtävät konkreettiset toimintansa kehittämisehdotukset.

nimi	**ongelmakohta**	**parannusehdotus**	**selitys/positiiviset seuraukset**
testaus-strategia	Tuotteen laatu ei parane, työskentely ei ole tavoitteellista.	Testausstrategian tai suunnitelman laatiminen kokonais- ja osatuotteille.	Laatutoimenpiteet toteutuvat suunnitellusti ja resurssien sisällä.
manuaalinen testaus	Manuaalinen testaus on liian harvan ihmisen harteilla, jolloin se on suppeaa ja itseään yksipuolisesti toistavaa.	Painopiste siirretään julkaisujen suuntaan. Mukaan otetaan laajemmin liiketoiminnan edustajia. Alfa- ja beeta-testaus otetaan tarkemmin hallintaan.	Kattavampi testaus tuo enemmän virheitä esiin. Käytettävyysasioita tulee esille ja tieto tuotteesta yrityksen sisällä lisääntyy.
automatisoitu testaus	Järjestelmätason testejä ei käytännössä ole, jolloin tuotekehitys on virhealtista ja tehotonta.	Automaattisten järjestelmätestien määrän lisääminen osana jokaista pyrähdystä.	Virheiden määrä vähenee ja uusien piirteiden lisääminen nopeutuu, eli tuotekehitys tehostuu.
Scrumin kehittäminen	Ei Scrum-mestaria, ei prosessin kehitystä. Itemien hyväksymiskriteerit ovat löysät. Kommukaatio virheistä yrityksen sisällä on heikkoa.	On valittava Scrum-mestari. On parannettava hyväksymiskriteerien seurantaa. ATDD on otettava käyttöön. Laatukommunikaatiota on lisättävä yrityksen sisällä.	Parantaa prosessia ja tuotteen laatua.

Taulukko 12: Yrityksen B kehittämiskohteet.

nro.	toimenpide
1.	Laadun ja laadun kehittämisen visio ja strategia pidemmällä aikavälillä (laatujärjestelmä).
2.	Nimetty operatiivinen laatuvastaava.
3.	Technical Writer (samalla myös käytettävyyden testaaja).
4.	Scrum Master: tiimin toiminnan valmentaja.
5.	Laatupalaverit: järjestelmällinen laadun seurannan ja kehittämisen työkalu.
6.	Talon sisäinen alfa-testaaminen ja testaussessiot.
7.	Testikattavuuden seuraaminen.
8.	Valmiin määritelmä käsitteenä: Milloin ohjelmointitehtävä on valmis? (Acceptance Criteria, Done-Done).
9.	Bugien raportoinnin ja seurannan järjestelmä (tiimi B).

Taulukko 13: Yrityksen B toimenpiteet kehittämisehdotusten johdosta.

6 Johtopäätökset tuloksista

Yritykset A ja B ovat liian kovassa paineessa uusien toiminnallisuuksien tuottamiseksi, jotta tiimit voisivat aidosti kantaa vastuuta tuotteidensa laadusta. Tiimien pitää ottaa pyrähdyksensä paremmin haltuun ja hankkia hyviä perusteita laatutoimenpiteiden tekemiseksi. Tiimien tulee voida toimia aidosti itseohjautuvina ja kaikkivoipina, mitä varten niillä tulee olla riittävät oikeudet ja resurssit. Hyvänä edistysaskeleena on yrityksen A laatima kehittämisprosessi. Kumpikin yritys tarvitsee laatusuunnitelman ja testausstrategian turvaamaan tuotteiden laadun järjestelmällisen kehittämisen. Yritys voi ottaa liiketoiminnallisen riskin hyväksymällä välttävänlaatuisen tuotteen, mutta tämä päätös tulee tehdä perustuen mahdollisimman tarkkaan arvioon tuotteen nykyisestä ja tulevasta laadusta. Päätöstä tehtäessä tulee huomioida laadun riskit ja vaikutukset liiketoimintaan sekä lyhyellä että pidemmällä tähtäimellä. Laatusuunnitelman ja testausstrategian tulee tiiviisti linkittyä yrityksen tuote- ja liiketoimintastrategioihin. Tämä vaatii ylemmältä johdolta painopisteen muuttamista ja laadun arvioimista osana muita liiketoiminnallisia tavoitteita.

6.1 Manuaalista ja automatisoitua testausta on suunnattava ja lisättävä

Testauskäytännöt ovat liian painottuneita manuaaliseen testaukseen, ja yksikkötestausta tehdään liian vähän. Kumpikin yritys kaipaa automatisoitua, synkronoitua ja monipuolista järjestelmätason testausta tukemaan päivittäistä työskentelyä. Erityisesti eri osajärjestelmien ja tiimien osatuotteiden yhteinen integrointiympäristö olisi tärkeä kehitysaskel. Uusien testausjärjestelmien rakentamiseen tulee käyttää resursseja, vaikka ne ovatkin hetkellisesti pois uusien piirteiden lisäämisestä. Tämä korvautuu kuitenkin, kun uusien piirteiden lisääminen nopeutuu. Myös testien ylläpitoon tulee varata riittävästi resursseja. Joitain merkkejä tästä tehostumisesta oli havaittavissa tiimi B:n kokemuksista.

Yksikkötestien määrän lisäämiseksi tulee asettaa tavoitteita ja niiden kirjoittamista osana päivittäistä kehitystyötä tulee seurata ja ohjata. Testaajien ja kehittäjien välistä roolijakoa tulee hämärtää tai purkaa kokonaan. Kehittäjien tulee osana työtään kirjoittaa monipuolisia automatisoituja testejä pelkkien yksikkötestien sijaan. Testaajien rooli tiimissä olisi testien kirjoittamista ohjaava ja tukeva.

Manuaalisen testauksen tarvetta tulee vähentää lisäämällä iteraatioiden aikaista automatisoitua testausta. Manuaalista testausta tulee kohdistaa enemmän julkaisujen suuntaan. Testien suorittajiksi ja suunnittelijoiksi on saatava enemmän käyttäjän tarpeita ja tuotteen kokonaisideaa hyvin ymmärtäviä ei-teknisiä ihmisiä. Näitä löytyy yritysten omasta henkilökunnasta tuotteen tukipalveluista, kouluttajista, asennus-insinööreistä ja jopa asiakkaiden organisaatioista. Manuaalisen testauksen ulkoistamistakin kannattaa selvittää. Hyväksymistestauksen eri vaiheita tulee nopeuttaa ja synkronoida Scrumin kanssa. Tuotteen julkaisujen välinen aika voidaan saada tämän avulla lyhyemmäksi.

6.2 Scrum-prosessia on ylläpidettävä ja kehitettävä jatkuvasti

Scrum-prosessia tulee noudattaa ja edelleen kehittää. Scrum-mestarin käyttö on oleellista. Sprinttien pituuden poikkeaminen totutusta kuukaudesta on hyvä asia, koska se osoittaa harkinnan käyttöä ja prosessin jatkuvaa kehittämistä eli jälkiagilismia. Hyväksymistestejä käytetään yrityksissä liian vähän. Hyväksymistestien kirjoittamiseen tai suunnitteluun voisivat osallistua yhdessä tuotteen omistaja, tes-

taaja ja kehittäjät. Uuden ohjelmistopiirteen valmiiksi merkitsemisen kriteereistä tulee pitää tiukemmin kiinni ja erityisesti tulee tarkastella sen testauksen ja dokumentaation riittävyyttä. Tiimien on aika ajoin hyvä käydä keskusteluja laadusta ja Scrum-prosessista. Asiakkaita tulee käyttää prosessissa ja yrittää vaalia tiimin yhteyttä heihin tapaamisilla ja muulla yhteydenpidolla.

6.3 Yritysten reagointi suosituksiin

Yritykset suhtautuivat tutkimuksen tuloksiin ja suosituksiin myönteisesti. Yritys B listasi suoraan suositusten läpikäyntipalaverissa kehittämisideoita omaan prosessiinsa. Yrityksen A suuremman organisaation johdosta toimenpiteiden miettiminen jakautui useamman eri toimijan kesken.

Yritys A:n edustajien mukaan jokaiseen esille nostettuun ongelmakohtaan ollaan puuttumassa ja asiat ovat kehittymässä parempaan suuntaan. Yrityksen kommentit on esitelty tarkemmin liitteessä 6. Tutkimuksen teon jälkeen on kehittynyt vahva eteenpäin menemisen tunnelma ja tutkimus on onnistunut peilaamaan tunnelmaa yrityksessä ennen uudistuksia. Uudistukset olivat jo tutkimuksen aikana työn alla yrityksen laatiman kehittämissuunnitelman muodossa. Yutkimus on voinut osaltaan vauhdittaa ja selkiyttää yrityksen kehittämisprosessia.

Yrityksen B toimittamasta toimenpidelistasta näkyy yrityksen hyväksyneen suositukset ja ryhtyneen niiden sekä tutkielman muiden havaintojen perusteella parantamaan toimintaansa. Erityisen tärkeä parannustoimenpide on laadun ja testauksen suunnittelu strategisella ja operatiivisella tasolla. Suunnitelmien toimeenpano ja säännöllinen seuranta operatiivisella tasolla on oleellista. Kehittämisehdotusten perusteella tutkimuksesta oli yritykselle hyötyä ja tutkimuksen havainnot toiminnan kehittämisen tarpeista olivat oikeita.

Kahdessa tutkitussa ohjelmistoyrityksessä reagointi annettuihin suosituksiin osoitti tarvetta testauksen kehittämiseen. Testausta ei ollut tunnistettu eri aikajänteiden mukaan tai sille ei ollut laadittu selkeää liiketoimintaperusteista tavoitetta.

6.4 Tutkimustulosten yhteenveto

Tutkimuksen tulokset olivat osittainen pettymys tutkimuskysymyksen selvittämisen suhteen. Hyviä kokonaisratkaisuja Scrumin ja testauksen yhteensovittamiseksi ei löytynyt.

Tutkimus tuotti selvän kuvan yritysten ongelmista. Yritykset eivät ole kyenneet tekemään suunnitelmallisia ja strategisia ratkaisuja testauksen suorittamiseksi. Yrityksiltä puuttui laajempi suunnitelmallinen näkemys testauksen ja laadun kehittämiseen. Uusien laatu- ja testaustoimenpiteiden tarvetta ei analysoitu järjestelmällisesti, olemassa olevien käyttöä ei kehitetty pitkäjänteisesti eikä yrityksillä ollut kokonaiskuvaa tarvittavien toimenpiteiden keskinäisistä suhteista. Tämä ei todennäköisesti johdu ketterien menetelmien käytöstä, vaan yritysten liiketoiminnan strategiasta. Tuotteen laatua ei nähty liiketoiminnan tarpeiden näkökulmasta, joten yrityksen johto, asiakas tai tuotteen omistaja eivät halunneet käyttää tuotekehityksen resursseja heille perustelemattomaan ja näennäisesti hyödyttömään asiaan.

Tutkimus ei päässyt analysoimaan kokemuksen, analysoinnin ja mittauksen avulla pitkäjänteisesti kehitettyjä testauksen toimintamalleja. Sen sijaan löytyi reaktiivisesti johdettuja käytäntöjä, tulipalojen sammuttamista ja kädestä suuhun elämistä. Puuttumaan jäi progressiivinen ote tuotteen laatuun, tuotekehitykseen ja niiden kautta tuotteen arvoon liiketoimintastrategisena välineenä. Tutkimus osoitti, että yrityksillä on paljon tekemistä perusasioiden kuntoon saattamiseksi ennen hienojakoisempia ja mielenkiintoisempia parannustoimia. Tutkimuksen suositusten huomioonottamisessa yritykset osoittivat suurta halua asioiden kehittämiseen, ja lähes kaikki yrityksille tehdyt suositukset on otettu käyttöön tai ongelmakohtiin on toisella tavoin puututtu.

7 Yhteenveto

Kirjallisuuskatsaus ei löytänyt eheää esitystä siitä, kuinka testaus Scrum-prosessimallissa tai ketterissä menetelmissä yleensä tulee järjestää. Testauksen ja laadunvarmistuksen vanhat tekniikat ovat pääosin päivittämättä ketterään maailmaan, joka on ainoastaan paikoittain täydentänyt aiempia malleja. Erityisen puutteellisia ovat pidemmän aikavälin toimintamallit ja testauksen erityisalueiden soveltaminen ketterässä projektissa. Kysymys eri asioiden suhteista jää epäselväksi: mitä kaikkea pitää uusia, muokata tai ottaa käyttöön sellaisenaan siirryttäessä laatuvarmistettuun Scrum-prosessiin?

Ketterien prosessimallien suosittelema TDD- ja ATDD-testaus on enemmän osa kehitysprosessia kuin testausta. Oikein käytettyinä näillä ja muilla ketterillä laatutekniikoilla voidaan merkittävästi parantaa tuotteen laatua. TDD ja ATDD ovat toiminnallista, automatisoitua, positiivista ja varmentavaa testausta. On todennäköistä, että projekti tarvitsee myös ei-toiminnallista, manuaalista, negatiivista ja tutkivaa testausta. Projektille laadittavien laatusuunnitelman ja testausstrategian tulee antaa selvät vastaukset kunkin tyyppisen testauksen tarpeesta ja määrästä.

Tutkimusosassa selvitettiin menetelmiä ja tekniikoita, joita Scrum-prosessia noudattava tiimi käyttää testaukseen. Tutkimuksessa selvisi käytössä olevien tekniikoiden toimivan ja vain joiltain osin kaipaavan tarkennuksia ja kehittämistä. Tutkimuksen suurin löytö oli kuitenkin, että yrityksiltä puuttui laajempi ja suunnitelmallinen näkemys testauksen ja laadun kehittämiseen. Uusien laatu- ja testaustoimenpiteiden tarvetta ei analysoitu järjestelmällisesti, olemassa olevien käyttöä ei kehitetty pitkäjänteisesti eikä yrityksillä ollut kokonaiskuvaa tarvittavien toimenpiteiden keskinäisistä suhteista. Lisäksi tutkimuksessa selvisi, etteivät tiimit kyenneet ottamaan vastuuta laadusta, koska laatuun liittyviä toimenpiteitä tehdään iteraatioissa liian vähän. Myös Scrum-prosessimallin noudattamisessa oli korjaamisen varaa. Yritykset osoittivat halua ja kykyä parantaa toimintaansa ongelmakohtien tunnistamisen jälkeen.

Tutkimuksen syventyminen vain kahteen yritykseen rajaa tuloksista johdettavia laajempia johtopäätöksiä. Lisätutkimusta aiheesta voisi tehdä toistamalla saman tutkimuksen paljon suuremmalla otoksella ja mahdollisesti keskittyen johonkin tiettyyn osa-alueeseen. Strategista suunnittelua voisi tutkia perehtyen laatusuunnitelmien, testausstrategioiden tai vastaavien olemassaoloon, laadintaan ja käyttöön. Scrum-prosessimallin käyttöä voisi tutkia tutustuen tiimien tapoihin muunnella sitä ja ana-

lysoida näiden muutosten vaikutusta ohjelmiston laatuun. Myös automatisoidun testauksen TDD:ta ja ATDD:ta monipuolisemmat ja täysin synkronoidut ratkaisut voivat olla palkitseva tutkimuskohde. Mielenkiintoista olisi myös tutkia yritysten A ja B laatu- ja testaustoimintaa vuoden tai kahden päästä. Tällöin voisi arvioida tehtyjen suositusten ja yritysten päätösten vaikutusta laatu- ja testaustoimintaan.

Lähteet

ASR02 Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. *VTT Publications 478*, 2002.

Ast05 Dave Astels, *A New Look at Test Driven Development*, 2005, http://blog.daveastels.com/files/BDD_Intro.pdf [21.2.2008].

Bac94 James Bach, Process Evolution in a Mad World. *Proceedings of the Seventh International Quality Week, Software Research*, USA, 1994.

Bac06 James Bach, *Manual Tests Cannot Be Automated*, http://www.satisfice.com/blog/archives/58 [18.8.2007].

Bac07 James Bach, *Exploratory Testing Explained*, http://www.satisfice.com/articles/et-article.pdf [13.7.2007].

Bec01 Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries Jon Kern, Brian Marick, Robert Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas, *Manifesto for Agile Software Development*, 2001, http://agilemanifesto.org/ [30.9.2007]

Bec00 Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

Bei90 Boris Beizer, *Software Testing Techniques*. Thompson Computer Press, 1990.

BWK05 Stefan Berner, Roland Weber and Rudolf K. Keller, Observations and lessons learned from automated testing. *Proceedings of the 27th international conference on Software engineering*, USA, 2005, pages 571-579.

Bin00 Robert Binder, *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison-Wesley, 2000.

CrH02 Lisa Crispin and Tip House, *Testing Extreme Programming*. Addison-Wesley, 2002.

Cri06a Lisa Crispin, *Hiring a tester with an agile attitude*. Better Software, Orange Park, Volume 8, Number 3 (March 2006), pages 16-18,37.

Cri06b Lisa Crispin, *Driving Software Quality: How Test-Driven Development Impacts Software Quality*. IEEE Software, Volume 23, Number 6 (November 2006), pages 70-71.

DeS00 Peter DeGrace and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions*. Prentice Hall, Yourdon Press, 1990.

Dav00 Graham Davis, Managing the Test Process. *Proceedings of the International Conference on Software Methods and Tools 2000*, Wollongong, NSW, Australia, pages 119-126. Computer Society Press, 2000.

Dob07 Jamie Dobson, Performance Testing on an Agile Project, *Proceedings of the conference on AGILE 2007*, Washington D.C., USA, August 2007, pages 351-358.

Els07 Amr Elssamadisy, *Patterns of Agile Practice Adoption: The Technical Cluster*. InfoQ, USA, 2007.

EnR07 Albert Enders and Dieter Rombach, *A Handbook of Software and Systems Engineering*. Addison-Wesley, 2003.

FrW93 Phyllis Frankl and Elaine Weyuker. *A formal analysis of the fault-detecting ability of testing methods*. IEEE Transactions on Software Engineering, volume 19, number 3 (March 1993), pages 202-213.

Fow99 Martin Fowler, *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

Fow01 Martin Fowler, *Continuous Integration*, 2001, http://www.martinfowler.com/articles/continuousIntegration.html [11.1.2008].

Gal07 Bob Galen, *You Can Take Advantage Of Its Benefits Without Bending Over Backward*. Software Test and Performance, Volume 4, Number 6 (June 2007), pages 14-20.

Gor06 Jason Gorman, *Post-Agilism . Beyond the Shock of the New*, 2006, http://www.parlezuml.com/blog/bblog/docs/Post-agilism.pdf [14.12.2007].

Gor07 Jason Gorman, *Post-Agilism Explained (Pretentiously)*, 2006, http://parlezuml.com/blog/?postid=407 [14.12.2007].

Gra92 Robert Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

Hie74 Peter Hiemann, A New Look at the Program Development Process, *Proceedings of the 4:th Informatik Symposium on Programming Methodology*. IBM Germany, Wildbad, September 25-27, 1974, pages 11-37.

HiC01 Jim Highsmith and Alistair Cockburn, Agile Software Development: The Business of Innovation. *Computer*, volume 34, number 9 (2001), pages 120-122.

Hum89 Watts Humprey, *Managing the Software Process*, Addison-Wesley, 1989.

HJW06 Chih-Wei Ho, Michael J. Johnson, Laurie Williams, and E. Michael Maximilien, On Agile Performance Requirements Specification and Testing, *Proceedings of the conference on AGILE 2006*, Minneapolis, USA, July 2006, pages 47-52.

HoK06 Antawan Holmes and Marc Kellogg, Automating Functional Tests Using Selenium, *Proceedings of the conference on AGILE 2006*, pages 270-275.

HoM96 Joseph R. Horgan and Aditya P. Mathur, Software testing and reliability, *Handbook of Software Reliability Engineering*. Los Alamitos, California, IEEE Computer Society Press, 1996, pages 531-566.

IRL05 Juha Itkonen, Kristian Rautiainen and Casper Lassenius, Towards Understanding Quality Assurance in Agile Software Development, *Proceedings of the International Conference on Agility 2005*, Denver, USA, July 2005, pages 201-207.

ItR05 Juha Itkonen and Kristian Rautiainen, Exploratory Testing: A Multiple Case Study, *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2005)*, Noosa Heads, Queensland, Australia, November 2005, pages 84-93.

IML07 Juha Itkonen, Mika V. Mäntylä and Casper Lassenius, Defect Detection Efficiency: Test Case Based vs. Exploratory Testing, *Proceedings of the*

first International Symposium on Empirical Software Engineering and Measurement, September 2007, pages 61-70.

Itk07 Personal communication with Juha Itkonen about the material of a new book, 2007-2008, to appear.

Kan95 Cem Kaner, *Software Negligence and Testing Coverage*. Software QA Quarterly, Volume 2, Number 2 (1995), page 18.

Kan97 Cem Kaner, Improving the Maintainability of Automated Test Suites. *Proceedings of the 10th International Software Quality Week*, San Francisco, CA, May 1997, page 28.

KaL00 Cem Kaner & Brian Lawrence, co-hosts, *Los Altos Workshop on Software Testing Number 9 (Grey Box Testing)*. Sunnyvale, CA, March 2000.

KBP02 Cem Kaner, James Bach and Bret Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*. John Wiley & Sons, 2002.

Kan04 Cem Kaner, *The Ongoing Revolution in Software Testing*, Software Test & Performance Conference, Baltimore, MD, December 7-9, 2004.

KLV05 Mehdi Kessis, Yves Ledru, Gérard Vandome, Testing and instrumentation: Experiences in coverage testing of a Java middleware. *Proceedings of the 5th International Workshop on Software Engineering and Middleware SEM '05*, ACM Press, September 2005, pages 39-45.

Koh06 Jonathan Kohl, *Post-Agilism: Process Skepticism*, 8.6.2006, http://www.kohl.ca/blog/archives/000166.html [14.12.2007].

Koh07a Jonathan Kohl, *Test-Driven Development and Exploratory Testing*, 9.4.2007, http://www.kohl.ca/blog/archives/000183.html [14.12.2007].

Koh06 Jonathan Kohl, *Post-Agilism Frequently Asked Questions*, 28.4.2007, http://www.kohl.ca/blog/archives/000184.html [14.12.2007].

Koh07b Jonathan Kohl, *Conventional Software Testing on a Scrum Team*, 30.6.2007, http://www.informit.com/articles/article.aspx?p=415981&rl=1 [14.12.2007].

Lyn03 James Lyndsay, A Positive View of Negative Testing. *STAREAST 2003: Proceedings of the Software Testing Analysis and Review Conference.* Orlando, USA, May 2003.

Mar99 Brian Marick, When should a Test Be Automated?, *STAREAST 1999: Proceedings of the Software Testing Analysis and Review Conference*, Orlando, USA, May 1999.

Mar99 Brian Marick, How to misuse code coverage, *Proceedings of the 16th International Conference on Testing Computer Software*, Washington, D.C., USA, June 1999.

Mar01 Brian Marick, *Agile Methods and Agile Testing.* Software Testing and Quality Engineering Magazine, Volume 3, Number 5 (2001).

Mar04 Brian Marick, *A Roadmap for Testing on an Agile Project.* Agile Times, Number 5 (2004).

Mar03 Robert Martin, *Aristotle's Error or Agile Smagile.*, 2.9.2003, http://weblogs.java.net/blog/rmartin/archive/2003/09 /aristotles_erro.html [14.12.2007].

MMC06 Grigori Melnik, Frank Maurer and Mike Chiasson, Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective, *Proceedings of the conference on AGILE 2006*, Minneapolis, USA, July 2006, pages 35-46.

Mye75 Glenford Myers, *Reliable software through composite design.* Petrocelli/Charter, 1975.

Mye04 Glenford Myers, *The Art of Software Testing, 2:nd edition.* John Wiley & Sons, 2004.

Par97 Vilfredo Pareto, *Cours d'économie politique*, Lausanne, Rouge, 1897.

PeY07 Mauro Pezzè and Michal Young, *Software testing and analysis.* Wiley, 2007.

Pre97 Roger Pressman, *Software Engineering, Practitioner's Approach, 4:th edition.* McGraw-Hill, 1997.

Pul06 Michael Puleio, How Not to do Agile Testing, *Proceedings of the conference on AGILE 2006*, Minneapolis, USA, July 2006, pages 305-314.

RaW06 Rudolf Ramler and Klaus Wolfmaier, Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. *Proceedings of the 2006 international workshop on Automation of software test*, Shanghai, China, 2006, pages 85-91.

Ran07 Juha Rantanen, *Acceptance Test-Driven Development with Keyword-Driven Test Automation Framework in an Agile Software Project*, Master's Thesis, Helsinki University of Technology, 2007.

Rau04 Kristian Rautiainen, *Cycles of Control: A Temporal Pacing Framework for Software Product Development Management.* Licentiate Thesis, Helsinki University of Technology, 2004.

RMM05 Kris Read, Grigori Melnik and Frank Maurer, Student Experiences with Executable Acceptance Testing, *Proceedings of the Agile Development Conference 2005*, Denver, USA, July 2005, pages 312-317.

Sch96 Ken Schwaber, *Controlled Chaos: Living on the Edge.* American Programmer, volume 9, number 5 (April 1996), pages 10-16.

Sch04 Ken Schwaber, *Agile project Management With Scrum.* Microsoft Press, 2004.

Smi07 Hubert Smits, *Scaling Agile Processes: Five Levels of Planning*, Agile Journal, 8.5.2007, http://www.agilejournal.com/articles/articles/scaling-agile-processes:-five-levels-of-planning.html [11.1.2007]

Som07 Ian Sommerville, *Software Engineering, Eight Edition*, Addison-Wesley, 2007.

SMC74 Wayne Stevens, Glenford Myers and Larry Constantine, Structured Design. *IBM Systems Journal*, 13 (2), 1974, pages 115-139.

Sum07 Megan Sumrell, From Waterfall to Agile - How does a QA Team Transition?, *Proceedings of the conference on AGILE 2007*, Washington D.C., USA, August 2007, pages 291-295.

TaN86 Hirotaka Takeuchi and Ikujiro Nonaka, *The New Product Development Game*. Harvard Business Review, Volume 64, 1986, pages 137-146

Wei71 Gerald Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

Wey88 Elaine Weyuker, *The evaluation of program-based software test data adequacy criteria*. Communications of the ACM, volume 31, Number 6 (June 1988), pages 668-675.

Liite 1. Sähköposti yrityksille

Hei,

Haluaisin tutkia ohjelmistojen testausta yrityksessänne graduani varten.

Miksi juuri te?
Lasse Koskela Reaktor Innovationista ja Kai Virkki Efectestä suosittelivat teitä hyvän Scrumin käyttönne vuoksi.

Mitä saatte?

- kuvauksen testauksestanne
- vertailun muihin Scrum-yrityksiin
- yrityskohtaisen raportin kehittämiskohteista

Tutkimuksen vaiheet:

1. 3X1 h henkilöhaastattelut yrityksessänne
2. tulosten analyysi, vertailu muihin yrityksiin ja tarvittava kirjallisuustutkimus
3. 1,5 h raportinpalautus-palaveri

Luottamus

- Yrityksiä käsitellään gradussa ainoastaan peitenimen (yritys A, yritys B, jne.) turvin.
- Hyväksytän teillä kaiken yritystänne koskevan aineiston ennen julkaisua.
- Raportti on vain omaa käyttöänne varten eikä julkinen osa gradua.
- Teemme halutessanne Non-disclosure agreementin (NDA).
- Gradu on puhtaasti akateeminen eikä liity työnantajaani.

Gradu ja minä.
olen siis tekemässä opinnäytetyötäni Helsingin Yliopiston Tietojenkäsittelytieteen laitokselle, ohjaajanani toimii Juha Taina. Olen opiskeluni ohella töissä Efectellä, vastaten testauksesta ja laadunvarmistuksesta. Olemme käyttäneet Scrumia jo 2,5

vuotta. Opinnäytetyöni eli graduni aihe on: "Kyselytutkimus testauksesta Scrum-prosessimallia käyttävissä ohjelmistoyrityksissä.". Luonnos gradustani ja suunnitelmistani löytyy osoitteesta: http://www.cs.helsinki.fi/u/mpori/gradu/gradu_pori.pdf. Otan tutkimukseen mukaan 4 yritystä.

Pääsen aloittamaan kanssanne silloin kuin teille sopii. Oletteko kiinnostuneita osallistumaan tutkimukseeni? Voitte olla yhteydessä sähköpostilla tai puhelimella, niin voimme puhua tarkemmin.

Ystävällisin terveisin,

Mikko Pöri
040-737 4087

LUONNOS YRITYKSEN RAPORTIKSI

1. yleistä
1.1. yrityksestä
1.2. tuotteesta testauksen kannalta

2. laadunvarmistuksen tavoitteet ja strategia
2.1. vertailu muihin yrityksiin ja kirjallisuuteen
2.2. parannusehdotukset tavoitteista ja strategiasta

3. testausprosessi
3.1. yleiskuvaus prosessista
3.1.1. vertailu muihin yrityksiin ja kirjallisuuteen
3.1.2. yksityiskohtaiset parannusehdotukset prosessista
3.2. käytetyt testausmenetelmät
3.2.1 vertailu muihin yrityksiin ja kirjallisuuteen
3.2.2 yksityiskohtaiset parannusehdotukset nykyisten menetelmien käytöstä ja uusista menetelmistä
3.3. muut huomiot

4. yhteenveto

5. Suositus tutustumisen arvoisista lähteistä

Liite 2. Peruskysymykset haastatteluihin

Yrityksissä haastateltavat henkilöt (n. 45 min / per) ja erityiset painopisteet ko. haastattelussa:

- Scrum-mestari (Scrum-prosessin laatu ja erikoisuudet, laadunvarmistus prosessin kannalta)
- tuotteen omistaja (laadunvarmistus pidemmällä tähtäimellä, oma rooli/vastuu laadunvarmistuksesta)
- testaaja/testauspäällikkö (testaajan rooli, oma työkuva, työtehtävät)
- kokenut kehittäjä (oma rooli laadunvarmistuksessa)

Lisäksi kaikilta yleiset kysymykset:

- Minkä tyyppisiä virheitä tuotteestanne löytyy? Mitkä ovat vakavimmat ja mitkä ovat kuormittavimmat? Onko testausta kohdennettu näihin virheisiin?
- Miten testaus on organisoitu? (erillinen testaaja, tiimi, kehittäjien rooli, tuotteen omistaja jne.)
- Millä tavoin testaus käytännössä tapahtuu? (kuinka usein, kuka tekee mitä, kenelle ja koska raportoidaan jne.)
- Minkälaisia erilaisia testausmuotoja on? (ensin omin sanoin, sitten pyydetään ruksittamaan listasta.)
- Mitä muita laadunvarmistusmenetelmiä kuin testaus? (katselmoinnit, XP-käytäntöjä: pariohjelmointi, TDD, refaktorointi, yhteisomistajuus, jatkuva integraatio)
- Miten hyvänä pidät yrityksen testausta? Mitä siinä on erityisen toimivaa?
- Mitä pitäisi parantaa/lisätä, mikä ei toimi tarpeeksi hyvin/ollenkaan?
- Onko testaus priorisoitu? Onko se priorisoitu mielestäsi oikein? Mitä pitäisit tärkeimpänä testauksen osa-alueena (yksi tai monta tärkeysjärjestyksessä)?

Yrityksiltä kerättävä materiaali:

- Yleiskuvaus testausprosessista
- Testausstrategiat, testaussuunnitelmat, testausraportit
- Kattavuusraportit ja analyysit eri testausmuodoista
- Yhteenvetoraportit ja analyysit virheiden määristä, sijainneista, vakavuuksista.

Työkalut ja tekniikat:

- Kehitystyökalut ja niiden laadunvarmistustoiminnot
- Käytettävät ohjelmointikielet, tekniikat, kirjastot pääpiirteissään. Teknologiaspesifinen laadunvarmistus
- Testaustyökalut: nauhoitus, testien kirjoitus, ympäristötyökalut, kattavuustyökalut

Liite 3. Sähköpostikysymys agile-testing@yahoogroups.com

Kysymys

From: agile-testing@yahoogroups.com [mailto:agile-testing@yahoogroups.com] On Behalf Of Mikko Pöri EFECTE
Sent: Sunday, January 13, 2008 7:03 AM
To: agile-testing@yahoogroups.com
Subject: Levels of agile planning and quality/testing?

Hello,
As defined nicely in this article:
http://www.agilejournal.com/articles/articles/scaling-agile-processes:
-five-levels-of-planning.html
there are five levels of agile planning:

- product vision
- product roadmap
- release plan
- iteration plan
- daily plan

As I see it, TDD, ATDD and other quality practises by the team answer the question for iteration and daily -levels, but leave the higher levels unexplained. So, what should the product vision, roadmap and release plan say about quality and at what level should quality goals or tasks be defined in these plans?

So, do you have any good articles or other resources in mind? Or how has your company addressed quality or testing in the vision, roadmap or release plan?

Yours,
Mikko Pöri

Vastaus 1

On Jan 14, 2008 11:17 AM, Bradley, Todd <todd.bradley@polycom.com> wrote:

At my company, the 'release plan' calls for a high level quality document, which I usually write. It talks about the general quality strategy, including stuff like test approach, test tools required, metrics, release criteria, who's responsible for what, and so on.

In rare cases, we've done a quality/testing document for a product roadmap, but it's so fuzzy as to be almost worthless.

With both of those documents, it's one of those things where you write it once, shop it around the organization, talk about it some, and check off a box on some corporate-specified product development checklist. And then by a month later everyone forgets the document ever existed.

Hope that helps,
Todd.

Vastaus 2

From: agile-testing@yahoogroups.com on behalf of John Overbaugh
john.overbaugh@gmail.com
Sent: 14. tammikuuta 2008 20:33
To: agile-testing@yahoogroups.com
Subject: Re: [agile-testing] RE: Levels of agile planning and quality/testing?

We're by no means experts here (still in the midst of our first agile project), but we're finding that the following materials are helpful to *us* in our project:

- Test strategy: overall strategy for testing, as Todd talks about below
- Cycle test plan: for the lack of a better noun... this doc just talks about what might be special in the current cycle and lists a series of milestones/tasks in the cycle
- Hardening test timeline: while we're agile, we are still taking 2-4 weeks between cycles to 'harden' (deeper integration-type test cases, perf test, etc.) and listing out the tasks we need to accomplish during that timeline. I suppose this could be done as workitems in the backlog...
- Acceptance tests: as was proposed on this list a while back, we're starting to add acceptance test cases to each user story. It's helping us in several ways: 1. we're more aware of what's happening, because we're looking at the user stories long before they're coded

2. we're 'linking' work between user stories (ie, we're able to test set/remove more efficiently because we look at them as a whole)
3. we're getting a better idea of the work up-front.

YMMV... the whole point of agile is that there arent' hard-and-fast rules that apply to each company or even each team. The goal is to decrease the size of a deliverable, drive the quality of deliverables up, and increase the frequency of releasing deliverables. The above are things we're finding that help us - you might or might not benefit in the same manner.

John O.

– John Overbaugh
blog: http://thougthsonqa.blogspot.com
MSN IM: john@overbaugh.com
Yahoo IM: johnoverbaugh

Liite 4. Yritys A, kehittämisehdotukset

Kehittämisehdotukset

Osatuotteista(tuote A ja sen sisartuotteet) koostuvan kokonaistuotteen nopeampi automaattinen rakentaminen ja asennus.

- Viimeisimmät lähdekoodit jokaisesta aliprojektista yhteisiin asennuksiin vasta kun tuote on paikallisesti rakentunut, suorittanut menestyksellisesti yksikkö- ja järjestelmätestit ja muut mahdolliset tarkistukset. Täten ei mikään tiimi joudu liikaa ylläpitämään muiden koodia.
- Edellä mainittuja automaattisia testejä on oltava riittävästi jokaisen tiimin tuotteelle.
- Testien ylläpitoon tulee varata resursseja osana pyrähdyksiä.
- Toimenpide pyrkii vähentämään merkittävästi integraatio-tiimin tekemää työtä eri tuotteiden integroimiseksi ja lyhentämään tuotteen julkaisuaikaa.
- Kunkin tuotetiimin tulee kirjoittaa automaattisia integraatiotestejä testaamaan erityisesti tuotteiden välisiä rajapintoja.
- Projektina suuri ja esimerkiksi ulkopuolista asiantuntija- ja konsulttiapua kannattaa käyttää.

Yksikkötestien kirjoittaminen ja refaktorointi osana jokaista pyrähdystä.

1. Priorisoitava jokainen moduuli/vastaavan kokoinen osa koodia.
2. Valittava jokaiselle moduulille eri kattavuustavoite riippuen moduulin kriittisyydestä ja virhealttiudesta.
3. Syötettävä sisään sprintteihin pienissä paketeissa.
4. Moduulille kirjoitetaan kattavat ja hyvät yksikkötestit sekä samalla refaktoroidaan.
5. Kattavuutta ja tavoitteiden onnistumista seurattava yhteisesti. Tuotteen laatu on koko tiimin vastuulla, ei vain testaajien tai tuotteen omistajan!

Automaattisten järjestelmätestien määrän lisääminen osana jokaista pyrähdystä.

- Priorisoitava tärkeimmät käyttötapaukset ja näille kirjoitettava automaattiset testit
- Myös kehittäjät kirjoittamaan automatisoituja järjestelmätestejä. Kehittäjien ja testaajien välistä rajaa kannattaisi madaltaa, ehkä samaten muitakin erikoistumisrooleja.
- Syötettävä sisään sprintteihin pienissä paketeissa.

Hyväksymiskriteerien tiukentaminen.

- Jokaisesta itemistä katselmoidaan pienessä ryhmässä (3-5 henkeä) ennen hyväksymistä dokumentointi ja testaus.
- Jokaiselle uudelle itemille on kirjoitettu hyväksymistesti(yksikkö, integraatio tai järjestelmä).

Testausstrategian tai suunnitelman laatiminen tuotteelle A.

- Pohjatiedoiksi tarkka analyysi bugeista, asiakkaan tukitarpeista ja kriittisistä liiketoimintariskeistä. Tulkittava asiantuntijoiden kanssa.
- Määritellään erityiset kehittämisen kohteet ja askeleet(keinot) kunkin toteuttamiseksi. Asioista on keskusteltava tarpeeksi ja huolellisesti!
- Valitaan niiden keskinäinen tärkeysjärjestys (joko yksi prioriteettijono, tai kaksi eli yksi tiimille ja toinen liiketoiminnalle).
- Selvitetään testauksen ulkoistamisen mahdollisuudet Suomeen tai ulkomaille.

Sprinttien lyhentäminen 1-2 viikkoon.

- Testaus pysyisi paremmin iteraation tehtävissä mukana.
- Hyväksymiskriteerit paremmin seurattavissa.
- Edellytyksenä maantieteellisen jaon lopettaminen ja tiimin keskittäminen yhteen paikkaan. Muuten palavereihin kuluva aika muodostuu suhteessa pyrähdyksen kestoon liian suureksi.

Liite 5. Yritys B, kehittämisehdotukset

Erityiset kehityksen kohteet:

Laadittava testausstrategia:

- Yhteistuote kokonaisuutena ja jokainen tuote erikseen
- Pohjatiedoiksi tarkka analyysi bugeista, asiakkaan tukitarpeista ja kriittisistä liiketoimintariskeistä. Tulkittava asiantuntijoiden kanssa.
- Määritellään erityiset kehittämisen kohteet ja askeleet(keinot) kunkin toteuttamiseksi. Asioista on keskusteltava tarpeeksi ja huolellisesti!
- Valitaan niiden keskinäinen tärkeysjärjestys (joko yksi prioriteettijono, tai kaksi eli yksi tiimille ja toinen liiketoiminnalle).
- Selvitetään testauksen ulkoistamisen mahdollisuudet Suomeen tai ulkomaille.

Testiautomatisaation lisääminen(XP!)

- Tuotteen, asennuksen ja laitteiden automaattinen testaus profiloiduilla käyttötapauksilla(tärkeimmät, yleisimmät) -> yksinkertaiset virheet pois, kattavampi regressio.
- Suorituskyvyn testaus(performance, stress, load).
- Asennus eri ympäristöihin automaattisesti.
- Testaajat ja kehittäjät kirjoittamaan testejä, varattava aikaa sprinteistä pienissä paloissa kehityskohteille.

Manuaalisen testauksen kehittäminen.

- Painopiste manuaalisessa testauksessa enemmän julkaisujen suuntaan.
- Mukaan enemmän liiketoimintapuolen ihmisiä, asennusinsinöörejä ja asiakkaita.
- Manuaalinen testaus suunnitelmallisemmaksi ja keskitetysti johdettavaksi(yksi ihminen, sama henkilö vastaa kaiken testauksen kehittämisestä).

- Alfa-testaus tutkivana testauksena tai suunnitelluilla käyttötapauksilla useilla työpareilla, joilta varataan sopivasti aikaa (esim. 2X2h) ja avuksi annetaan yksi testaaja(esittelee menetelmän, pitää flown yllä, kirjaa ylös käytettävyyshuomiot ja virheet).

- Beeta-testaus ohjatummaksi, testaaja yms. insinöörin mukaan asiakkaalle selvän listan kanssa, testausta yhdessä asiakkaan kanssa, selvät raportit jokaisesta. Webbi-lomake on hyvä idea!

Scrumin kehittäminen.

- Scrum-mestarin käyttöä ehkä harkittava, rooli on kuitenkin valmentava ja sillä on merkitys. Roolia ei pitäisi sekoittaa muihin rooleihin, kuten tiimin vetäjän, testaajan, tuotekehityksen johtajan, tuotteen omistajan tai muihin vastaaviin.

- ATDD:n lisääminen, tutkittava uusia keinoja ja tekniikoita kirjoittaa automaattisia hyväksymistestejä esim. tuotteen omistajan, testaajan tai kehittäjän toimesta

- Lisättävä itemien(korttien) valmistumisehtoihin tarkistukset dokumentaation ja testauksen tekemisestä, ehtoja noudatettava tiukemmin. Dokumentaation ja testauksen säännölliset katselmoinnit osana pyrähdystä.

- Kommunikaation lisääminen ja tiimin laatuvastuun vahvistaminen, laadusta keskusteltava myös pidemmällä tähtäimellä. Mikä on laadun (virheiden, korjausten, uusien ohjelmistopiirteiden) kommunikaation tilanne laajemmin, esim. asennusinsinöörien suuntaan?

Käytettävyystestausta ja konsultointia käyttöliittymän uusiutuessa riskin pienentämiseksi.

Asennustestaus ja kestävyystestaus eri käyttöjärjestelmillä ja erilaisilla laitteilla.

- Automatisoitava

- Käynnissä pitkän aikaa, testaajan esimerkin pohjalta

Liite 6. Yritys A palaute kehittämisideoista

Kysymys tutkimuksen suosituksista ja johtopäätöksistä

Terve,

kysynkin sitten teiltä: aiheuttiko työni tai suositukseni teille tai tiimille jotain muutoksia/parannusajatuksia? Vai olivatko johtopäätökset metsässä tai liian idealistisia?

Kiva olisi saada vastaus vielä tähän graduani ajatellen. Oli vastaus mikä hyvänsä, on se hyväksi gradulleni.

Kiitos.

Mikko

Laatupäällikön vastaus

Moi,

Sain työstäsi yhden todo:n, eli company level test strategy.

Näillä näkyvin laadin sen heinäkuussa ja esittelen meidän managereille elo-syyskuussa.

A tiimiltä kannattaa varmaan myös kysellä mitä ideoita he työstäsi saivat.

Tiimin vastaus

Moro!

Itse asiassa meillä on tilanne parantunut aika lailla. XXX poisti käytännössä kaiken kehityspaineen meidän niskasta nähtyään Jeff Sutherlandin puhumassa XXX XXX. Tavoitteena on nyt saada seuraava versio julkaistua ennen kuin siirrytään jatkokehittämään järjestelmää oikein toden teolla.

Lisäksi olemme saaneet järjestelmätestauksen automatisoinnin hyvään vauhtiin ja meillä on viitisenkymmentä testitapausta, jotka suoritetaan jokaisen uuden buildin yhteydessä. Jatkossa on tarkoitus kirjoittaa myös automaattiset testitapaukset jokaisen uuden toiminnon toteuttamisen yhteydessä.

Cross-functional tiimityöskentelyä on myös suunniteltu parannettavan tulevaisuudessa. Sen toteuttaminen vaan on melko hankalaa jo maantieteellisten seikkojen takia. Toinen ongelma vaikuttaisi olevan tietynlainen muutosvastarinta. Ihmiset kun ovat tottuneet hoitamaan sen 'oman ruutunsa'. Itse ainakin olisin valmis opettelemaan koodauksen jalon taidon, jos vaan asiantuntevaa ohjausta olisi tarjolla.

Product backlogin ylläpitämiseen on myös kiinnitetty aiempaa enemmän huomiota.

Meidän product owner priorisoi jatkuvasti backlogissa olevia itemejä ja kerran viikossa pidetään puolisen tuntia kestävä tilaisuus, jossa arvioidaan backlogissa olevien itemien vaatima työmäärä. Tämä vähentää selvästi varsinaiseen suunnittelupalaveriin vaadittavaa aikaa.

Yleisesti koko firmaa koskien voin todeta, että lähes kaikkiin esitelmässäsi mainitsemiisi ongelmakohtiin ollaan puuttumassa tulevaisuudessa. Olemme juuri aloittaneet koko firman kattavat synkronoidut pyrähdykset, joissa jokaisen kehitystiimin pyrähdyksen kesto on 3 viikkoa. Sprintin jälkeen pidetään stabilisointi pyrähdys, jonka tarkoituksena on varmistaa sujuva tuotantoon vienti. Tavoitteeksi on asetettu uuden version julkaiseminen jokaisen pyrähdyksen jälkeen.

Siinäpä ne tärkeimmät taisivatkin olla... eteenpäin ollaan menossa.

Liite 7. Yritys B palaute kehittämisideoista

Teknisen johtajan vastaus

Keskustelussa esille nousseet parannusehdotukset jotka kehittävät XX:n tuotekehityksen laatua:

1) Laadun ja laadun kehittämisen visio ja strategia pidemmällä aikavälillä (laatujärjestelmä)
2) Nimetty operatiivinen laatuvastaava
3) Technical Writer (samalla myös käytettävyyden testaaja)
4) Scrum Master: tiimin toiminnan coachaaja
5) Laatupalaverit: järjestelmällinen laadun seurannan ja kehittämisen työkalu
6) Talon sisäinen alpha-testaaminen ja testaus-sessiot
7) Testikattavuuden seuraaminen
8) Valmis käsitteenä: Milloin ohjelmointitehtävä on valmis? (Acceptance Criteria, Done-Done)
9) Bugien raportoinnin ja seurannan järjestelmä (XX-tiimi)